iHadoop: Asynchronous Iterations Support for MapReduce

Thesis by
Eslam Elnikety

Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science

King Abdullah University of Science and Technology
Thuwal, Makkah Province, Kingdom of Saudi Arabia

August, 2011

The thesis of Eslam Elnikety is approved by the examination committee.

Committee Member:	Panos Kalnis
Committee Member:	Sambit Sahu
Committee Chairperson:	Hany E. Ramadan

King Abdullah University of Science and Technology

2011

ABSTRACT

iHadoop: Asynchronous Iterations Support for MapReduce

Eslam Elnikety

MapReduce is a distributed programming framework designed to ease the development of scalable data-intensive applications for large clusters of commodity machines. Most machine learning and data mining applications involve *iterative* computations over large datasets, such as the Web hyperlink structures and social network graphs. Yet, the MapReduce model does not efficiently support this important class of applications. The architecture of MapReduce, most critically its dataflow techniques and task scheduling, is completely unaware of the nature of iterative applications; tasks are scheduled according to a policy that optimizes the execution for a single iteration which wastes bandwidth, I/O, and CPU cycles when compared with an optimal execution for a consecutive set of iterations.

This work presents iHadoop, a modified MapReduce model, and an associated implementation, optimized for iterative computations. The iHadoop model schedules iterations asynchronously. It connects the output of one iteration to the next, allowing both to process their data concurrently. iHadoop's task scheduler exploits inter-iteration data locality by scheduling tasks that exhibit a producer/consumer relation on the same physical machine allowing a fast local data transfer. For those iterative applications that require satisfying certain criteria before termination, iHadoop runs

the check concurrently during the execution of the subsequent iteration to further reduce the application's latency.

This thesis also describes our implementation of the iHadoop model, and evaluates its performance against Hadoop, the widely used open source implementation of MapReduce. Experiments using different data analysis applications over real-world and synthetic datasets show that iHadoop performs better than Hadoop for iterative algorithms, reducing execution time of iterative applications by 25% on average. Furthermore, integrating iHadoop with HaLoop, a variant Hadoop implementation that caches invariant data between iterations, reduces execution time by 38% on average.

ACKNOWLEDGMENTS

I was fortunate enough to receive valuable advice and support from many people through my graduate career, and for that I am thankful. Foremost, I am very grateful to my thesis advisor, Hany Ramadan, who has been dedicated to his students and to the research of our group. We had many discussions in different research topics, and his comments always put me in the right direction.

The ideas I present in this thesis were developed by many brainstorming sessions with Tamer Elsayed and Hany Ramadan. They offered many insightful comments regarding the design and the implementation of the system presented in this work.

I would like to thank Panos Kalnis and Sambit Sahu for their comments and suggestions that improved the quality and the presentation of this thesis.

I would also like to thank the members of the Advanced Systems Group for the fruitful discussions we had during our weekly pizza night.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter I

Introduction

We live in a digital world where data sizes are increasing exponentially. By 2007, the digital universe was estimated at 281 exabytes, and it is expected to experience more than a tenfold growth by 2011 [15]. Data are being generated at unprecedented scale and rate. This has a direct effect on the scale of datasets that need to be processed in several domains, with volumes varying from many terabytes to a few petabytes. For example, analyzing the hyperlink structure of the Web requires processing billions of Web pages, mining popular social networks involves millions of nodes and billions of edges, multi-dimensional astronomical data are collected at rates exceeding 100 GB/day [27], and Facebook is hosting over 10 billion photos.

I.1 Data Analysis Approaches

This explosion of available data has motivated the design of many scalable distributed frameworks [21, 11] to face the challenges presented when processing and analyzing those sizes. The key challenges every distributed framework has to target for large scale processing are:

- **Storing the data.** Data are usually stored in a distributed manner across the

disks of the computing platform. Techniques such as replication and striping are used to provide higher availability and higher aggregate disk throughput.

- **Distributing the load.** The frameworks have to select an execution plan to balance the work assigned to each node. This execution plan can be determined statically or dynamically, where the later is more likely to cope with fluctuations of the workloads presented to the system [34].

- **Exploiting parallelization.** It is up to the framework to manage and exploit the tasks that can be done in parallel. Some models partition the data into parts where each can be processed in parallel, i.e., partitioned parallelism. Others, e.g., stream databases [39], have several pipelines of execution running at the same time.

- **Providing fault tolerance.** Fault tolerance is critical for those architectures based on commodity machines where failures are very frequent, especially if the framework will be used for long-running workloads. Others based on expensive and specialized hardware can have weaker fault tolerance models customized for their needs and practices.

- **Programming model.** The model should be generic enough to express a large class of problems. Furthermore, frameworks try to provide an easy-to-use API for developers.

Other challenges are energy efficiency [2] and cost of ownership. Distributed and parallel data analysis frameworks have several common traits, but most notably, they aggregate computing power from the available computing infrastructure to perform large scale data processing.

One approach to large scale processing is to use high-performance expensive computing platforms, e.g., parallel databases and grid computing. *Parallel databases* are

robust high-performance computing platforms. They provide a high level programming environment, e.g., SQL, on top of a set of high-end servers. Parallel databases partition structured data according to a variety of partitioning strategies to achieve as much parallelism as possible when answering queries. Some parallel databases [12] assume shared-nothing nodes over high-speed interconnect. Those parallel databases have proven linear scalability, i.e., maintaining a constant response time as data sizes increase by proportionally adding nodes to the system, over clusters of order of 100 nodes. Yet, parallel databases are still expensive and not as scalable as necessary. Whereas, *high performance computing* and *grid computing* communities perform large scale processing using message passing programming paradigms [17], e.g., MPI libraries. The work is distributed and parallelized over a cluster of machines that share the same file system over a Storage Area Network (SAN). Grid computing is effective for compute-intensive workloads, e.g., scientific computations. For data-intensive workloads, this model is inadequate as I/O becomes dominant.

The other approach to large scale processing is to use clusters of commodity machines. MapReduce is a large scale data-intensive processing framework that scales out to thousands of commodity machines [11]. With MapReduce, developers can focus on their domain-specific tasks, while the framework is responsible for low-level system issues such as job scheduling, load balancing, synchronization, and fault tolerance which is critical for long-running jobs. Developers need to implement two functions, *map* and *reduce*, and do not need to be concerned about the failures of unreliable commodity machines, or complicated parallelism and synchronization constructs. MapReduce is used in many applications such as indexing [25], data mining [33, 23], and machine learning [7, 42].

Compared to other approaches, MapReduce is more scalable and more suitable for data-intensive processing [9]. Its ease of use is best reflected by its increasing popularity; Hadoop [18], an open source implementation of MapReduce, has been adopted by

several enterprises such as Yahoo!, eBay, and Facebook [41]. It follows the basic architecture and programming model initially introduced by Google's MapReduce [11].

I.2 Iterative Applications

Iterative computations represent a large and important class of applications. They are at the core of several data analysis applications such as PageRank [3], k-means, Hyperlink-Induced Topic Search (HITS) [24], and numerous other machine learning and graph mining algorithms [35, 49, 38]. These applications process data iteratively for a pre-specified number of iterations or until a termination condition is satisfied. Figure I.1 depicts such behaviour. The output of an iteration can be used as input to subsequent iterations.

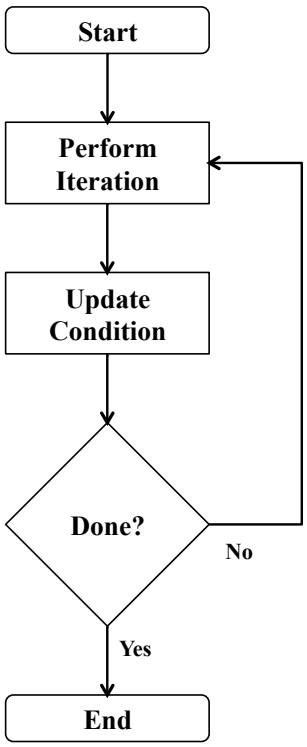


Figure I.1: Flow chart for an iterative application execution. `Done?` returns `Yes` whenever the execution (1) finishes running the pre-specified number of iterations, or (2) meets the termination conditions.

I.2.1 The Need for Scalable Iterative Solutions

Given the massive sizes of data and the importance of iterative computations, we need a scalable solution to efficiently apply iterative computations on large scale datasets. If we are to run the PageRank algorithm on Internet-scale datasets, we have to process iteratively over 1 billion Web pages; whereas mining the Internet will require processing 25 terabytes of data in every iteration[1].

On one hand, the MapReduce framework scales to thousands of machines, but on the other hand, it was not designed to run iterative applications efficiently. Since a series of iterations can be represented as two or more back-to-back MapReduce jobs, developers can use a driver program to submit the necessary MapReduce jobs for each iteration. If the application has to meet certain criteria before stopping, the driver program is responsible for running the termination check. The termination check itself can be a separate MapReduce job, or a separate program that is invoked by the driver.

This approach to using iterative algorithms with MapReduce has two significant limitations. An iteration must wait until:

- the previous iteration has finished completely and its output has been entirely written and committed to the underlying file system, and
- the termination check, if needed, has finished.

Since each iteration starts, usually, by reading from the file system what has just been written by the previous one, significant amounts of precious network bandwidth, I/O, and CPU cycles are wasted. The work presented in this thesis targets these limitations and provides a modified MapReduce model that can run iterative applications efficiently.

[1]These numbers are based on the ClueWeb09 dataset `http://lemurproject.org/clueweb09.php/`

I.3 Contributions

This work presents *iHadoop*, a modification of the MapReduce framework optimized for MapReduce applications that require iterative computations. iHadoop modifies the dataflow techniques and task scheduling of the traditional MapReduce model, to make them aware of the nature of iterative computations. iHadoop strives for better performance by executing iterations *asynchronously*, where an iteration starts before its preceding iteration finishes. This is achieved by feeding the output of an iteration as it progresses to the following one which allows both to process their data concurrently. Extra care is taken to preserve the MapReduce design principle of transparent fault tolerance, even with asynchronous iterations. iHadoop also modifies the task scheduler so as to schedule tasks that exhibit a producer/consumer relation of consecutive iterations on the same physical node *where* the cost of inter-iteration data transfer is minimal. Since iterations are running asynchronously with iHadoop, its task scheduler decides *when* it is optimal to schedule a certain task.

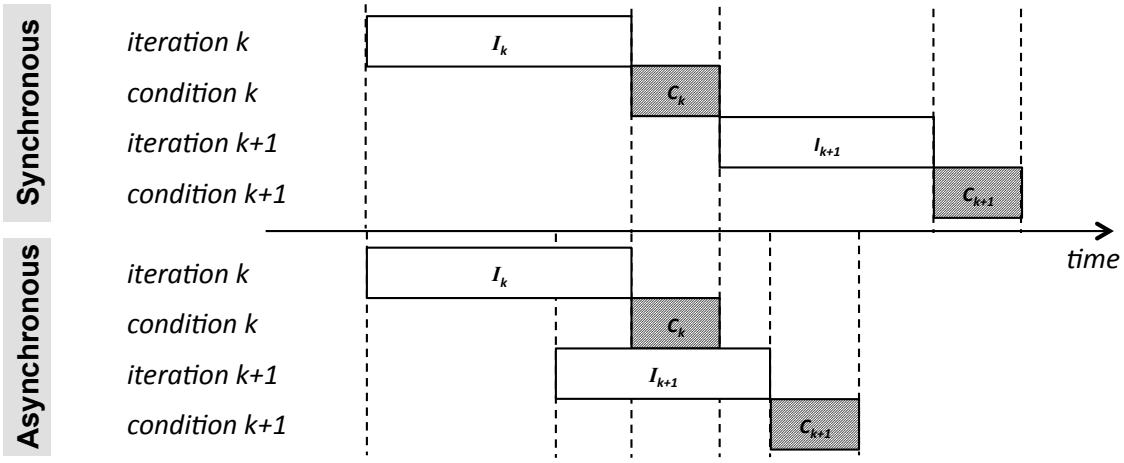


Figure I.2: Synchronous iterations versus asynchronous iterations for an iterative application.

Some applications need to check the termination condition between every two consecutive iterations I_k and I_{k+1}. If the application converges after n iterations,

the termination condition is satisfied only once out of n times. iHadoop runs this termination check in the background after starting iteration I_{k+1} asynchronously, speculating that the condition will not be satisfied after the completion of iteration I_k. If it turns out to be satisfied, iteration I_{k+1} is immediately terminated; this results in wasted computation, but it is insignificant compared to the performance gains achieved by asynchronous iterations. If the condition is not satisfied, I_{k+1} will have made considerable progress by the time the check concludes as we show experimentally in Chapter VI. Figure I.2 visually shows the difference between the execution pipelines of synchronous and asynchronous iterations, in the presence of termination checks.

This work makes the following main contributions:

- It introduces iHadoop, a modified version of the MapReduce model that allows iterations to run asynchronously. iHadoop also schedules the termination check, if needed, concurrently with the subsequent iteration. This results in greater parallelism at the application level and better utilization of resources.

- It modifies the MapReduce task scheduler to be aware of the asynchronous behavior. iHadoop takes advantage of inter-iteration locality and schedules the tasks that have strong dataflow interaction on the same physical node whenever possible.

- It presents an associated implementation of the iHadoop model, and describes issues encountered while tuning the performance of asynchronous iterations.

- It presents an evaluation of iHadoop along with a performance comparison with Hadoop, using real and synthetic datasets.

I.4 Walkthrough of this Thesis

The rest of this thesis is organized as follows. Chapter II gives an overview of the MapReduce framework. Usage of commodity hardware, automatic parallelization, and scalability are the key principles of the MapReduce model. The MapReduce programming model has only two primitives: *map* and *reduce*. However, this model is general enough to represent a very broad class of applications. This chapter also illustrates how iterative applications are executed with the traditional MapReduce framework—readers familiar with MapReduce can skip this chapter.

Chapter III discusses the related work. Section III.1 describes the MapReduce implementations that focus on optimizing the framework for iterative computations by modifying the underlying system. Section III.2 describes those systems that provide support for a pipeline of MapReduce jobs by adding an interface to existing MapReduce frameworks. Section III.3 discusses other alternatives to MapReduce.

Chapter IV presents the design of iHadoop. It examines the different types of dependencies that arise in a set of consecutive iterations in Section IV.1. Section IV.2 argues that the data dependency found between two consecutive iterations, where the output of an iterations is used as input to the next one, can be carefully handled to give room for executing those two iterations asynchronously. Section IV.3 examines how a control dependency between an iteration and the following termination check can be ignored to further reduce the application's latency. Section IV.4 describes the task scheduling policy required by the iHadoop model to support asynchronous iterations. Sections IV.5 and IV.6 discuss how the fault tolerance and load balancing mechanisms of iHadoop are as efficient as those of MapReduce.

Chapter V introduces our implementation of the iHadoop model. It describes in detail the implementation decisions required to tune the performance of iHadoop. This chapter answers questions such as: When should iHadoop schedule reducers of the following iteration? How does iHadoop transfer the data between the tasks

that exhibit a producer/consumer relationship? And how does iHadoop control the dataflow from a reducer to the mappers of the following iteration with minimal overhead?

Chapter VI evaluates the performance of our iHadoop implementation. It starts by describing the experimental setup and the applications it will use through the evaluation in Sections VI.1–VI.2. Section VI.3 evaluates the performance gains achieved through asynchronous iterations and running the termination condition concurrently with the subsequent iteration, speculatively. Since iHadoop takes advantage of other optimizations proposed for iterative applications over MapReduce (such as caching invariant data between iterations [4, 14]), this chapter analyzes the performance gains of different optimizations available to iHadoop. Results show that asynchronous iterations improve the performance of the underlying system, on average, by 20% regardless of the other optimizations implemented by the underlying system.

Chapter VII concludes this thesis with some insights on the future work.

Chapter II

MapReduce Framework

MapReduce, introduced by Dean and Ghemawat in 2004, is a framework for large scale data processing using commodity clusters [11]. While originally used for processing Web search-related data, its simple programming interface is flexible enough to be used by a broad range of applications. The framework transparently distributes data, allocates tasks, and parallelizes computations across cluster nodes in a shared-nothing architecture. It is extensively used in large scale data centers such as those operated by Google, Facebook, and Amazon.com. This chapter presents a brief overview of the MapReduce model.

II.1 Principles

This section introduces some of the key principles underlying the design of the MapReduce framework.

- **Commodity Hardware.** MapReduce does not require expensive hardware or special network and storage subsystems. The model assumes a large cluster of commodity machines connected through a local network. Every machine in the cluster is assumed to have its local storage. The framework makes no assump-

tions regarding performance homogeneity across nodes. Clusters designed to run MapReduce should be cost-effective to purchase due to the use of inexpensive hardware, and the availability of open-source implementations.

- **Automatic Parallelization.** The MapReduce framework automatically handles many system-level issues; it automatically splits input data, balances the load across nodes, transfers data across the network, synchronizes jobs, handles node failures, and so on. This allows developers to focus on solving their problems rather than dealing with the challenges of distributed programming, thus increasing overall productivity.

- **Scalability.** MapReduce uses a shared-nothing architecture which is more scalable than shared-disk or shared-memory architectures. MapReduces scales up to thousands of machines and can process many Terabytes of data. MapReduce is able to recover from faults that occur during job execution due to its sophisticated fault tolerance model.

II.2 Programming Model

The programming model adopted by MapReduce is inspired by functional programming. The framework defines two primitives to be implemented by the user: *map* and *reduce*. These two primitives have the following signatures:

$$map: \ (k_1, v_1) \rightarrow [(k_2, v_2)]$$

$$reduce: \ (k_2, [v_2]) \rightarrow [(k_3, v_3)]$$

The *map* function is applied on every (k_1, v_1) pair and produces a list of intermediate (k_2, v_2) pairs. The *reduce* function is applied on all intermediate pairs with the same key and outputs a list of (k_3, v_3) pairs.

II.3 Runtime System

A sample runtime execution is shown in Fig. II.1. In the Map phase, the runtime splits the input into chunks. The set of map tasks, operating in parallel across different nodes, process the chunks by applying the *map* function on the (k_1, v_1) pairs in their input splits. In the subsequent Shuffle phase, the resultant intermediate (k_2, v_2) pairs are grouped and partitioned by the runtime according to the key k_2 across the cluster nodes. Lastly, in the Reduce phase, those partitions are processed in parallel by reduce tasks, each of which applies the *reduce* function to its partition to produce a list of (k_3, v_3) pairs as the job output.

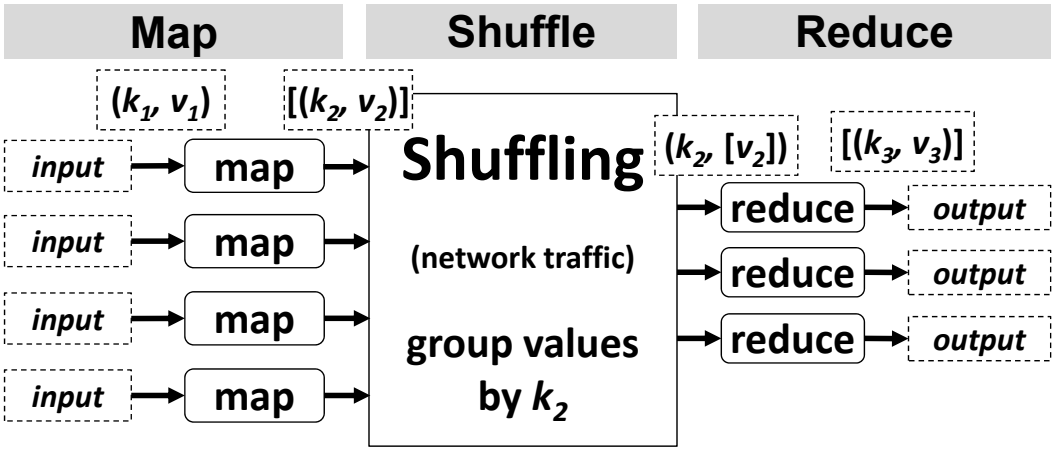


Figure II.1: Overview of the MapReduce framework. Each map task processes an input split from the distributed file system and creates an output partition for each reduce task. After performing remote reads to retrieve their partitions (from the map tasks), the reduce tasks sort, group and process their input, and then write their output to the distributed file system.

The MapReduce runtime is responsible for several key decisions such as scheduling tasks across cluster nodes. Since MapReduce runs on a large number of commodity machines, which are not assumed to be very reliable, the runtime provides fault tolerance. MapReduce does not assign tasks to nodes statically before execution; instead, task scheduling is dynamic, making it easier to handle node failures by reassigning

failed tasks to other working nodes.

II.4 Iterative Applications and MapReduce

As discussed in Subsection I.2.1, developers can use MapReduce framework to run iterative applications using a driver program. Figure II.2 illustrates the interaction between this driver program and the MapReduce framework.

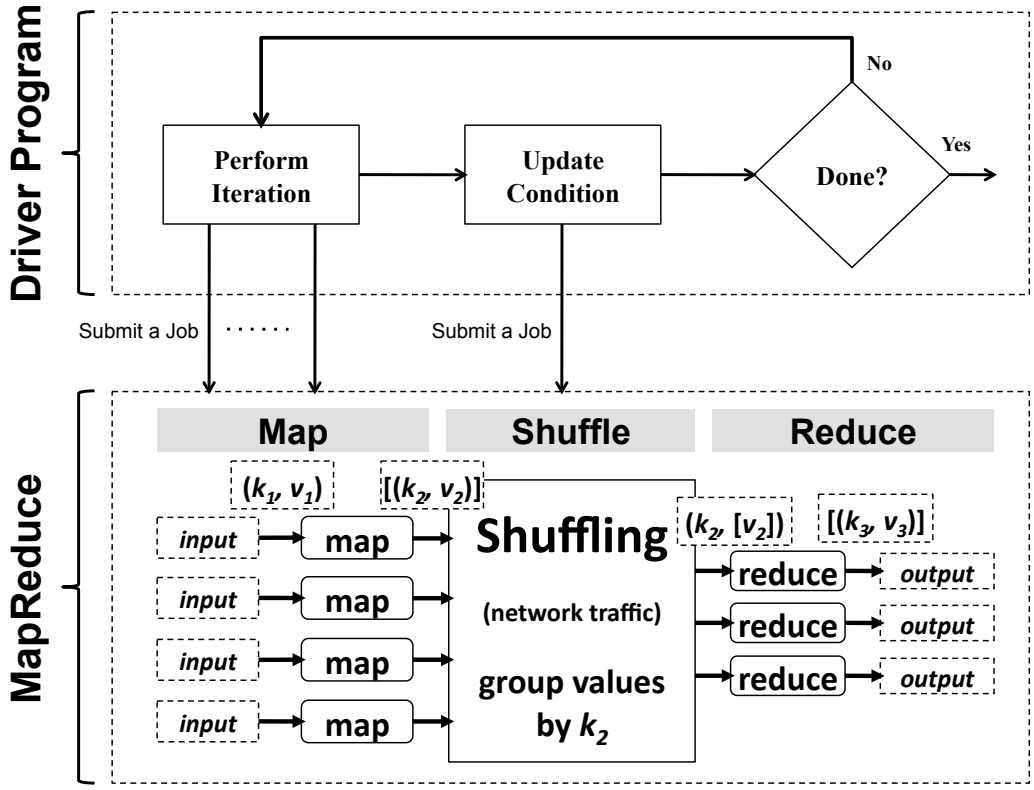


Figure II.2: Typical iterative MapReduce application. Each iteration of the application is one or more MapReduce jobs. After each iteration, the driver program submits a termination check job.

Each iteration of the application is assumed to fit into one or more successive MapReduce jobs (step). The driver program is responsible for configuring each step: adjusting input and output directories and providing the appropriate *map* and *reduce* functions that corresponds to the running step. After successfully submitting all the

steps that constitute a complete iteration, the driver program runs the termination check if the application requires. The termination check, which can be a MapReduce job or a separate program, updates the termination condition which is finally checked by the driver program to determine if the application has satisfied its termination criteria and can stop.

Chapter III

Related Work

This chapter reviews related models and systems. We first describe several implementations of the MapReduce model described in Chapter II, followed by higher-level systems that are built on top of MapReduce. Finally, we review frameworks other than MapReduce that are also designed for large scale data processing.

III.1 Iterative MapReduce Implementations

There are many implementations of the MapReduce model [18, 10, 45, 30, 19, 44, 37, 6]. Apache Hadoop [18] is an open-source Java implementation of MapReduce. Its main components are the Hadoop Distributed File System (HDFS), an implementation of the Google File System (GFS) [16], and the Hadoop MapReduce component, an implementation of Google's MapReduce [11]. HDFS is a block-based distributed file system that seamlessly partitions and replicates data across cluster nodes. Hadoop has a master node and many worker nodes. The master node runs a `JobTracker` process which is responsible for scheduling tasks on worker nodes according to the policy implemented by its `TaskScheduler`. The `JobTracker` keeps track of finished and in-progress tasks, it also orchestrates speculative execution and fault recovery. Each worker runs a `TaskTracker` process which receives a periodic `heartbeat` from the

`JobTracker` to launch map and reduce tasks locally on its node. The `TaskTracker` replies back with information about its load, available resources, and the progress of its tasks. It also handles requests for intermediate output of map tasks. Hadoop does not have explicit support for iterative applications. Iterative MapReduce applications on Hadoop will follow the design illustrated in Figure II.2, where the MapReduce framework is completely unaware of the iterative behaviour of those applications, and will suffer from the limitations discussed earlier in Subsection I.2.1. The following describes the MapReduce systems that optimize the framework for iterative applications.

III.1.1 HaLoop

HaLoop [4] is a modified version of Hadoop that supports iterative large scale applications. It provides a programming API to express iterative applications. For most of iterative applications, a large part of the input data remains unchanged from one iteration to the next. HaLoop caches and indexes loop-invariant data for map and reduce tasks on the local disks of the nodes, for iterative applications that follow this construct:

$$R_{i+1} = R_0 \cup (R_i \bowtie L)$$

where R_0 is the initial result and L is an invariant relation. And to take advantage of caching, HaLoop modifies the task scheduler to reassign to each node of the cluster the set of tasks that were assigned to it in earlier iterations. This reduces significantly the amount of data that need to be shuffled between the map and the reduce tasks of the same iteration. For those iterative applications that require a termination check, HaLoop introduces *fixpoint evaluation* by comparing, in a distributed manner, the current iteration output with the output of the previous iteration. To do this efficiently, HaLoop caches and indexes the most recent reducer output locally on each

reducer node. This significantly reduces the time needed to evaluate the termination condition. However, this evaluation is limited only to those iterative algorithms where the convergence criteria depend only on the output of the two most recent iterations.

III.1.2 Twister

Twister [14] is a stream-based MapReduce implementation that supports iterative algorithms. It uses a publish/subscribe messaging infrastructure configured as a broker network for sending/receiving data to/from tasks and daemons. Twister as well distinguishes between static data, which are these data that remain fixed throughout the whole computation, and dynamic data, which are the results of the computation, in an iterative application. Twister configures the map and reduce tasks to load the static data whenever they start. And to avoid repeatedly reloading the static data in every iteration, Twister uses long running map/reduce tasks, i.e., Twister does not initiate new map and reduce tasks for every iteration. In case of failures, the entire computation is rolled back few iterations to the last saved state. To take advantage of the long running tasks and cached data, twister has to use static scheduling to assign map and reduce tasks to fixed locations which might lead to unoptimized resource utilization and load unbalancing. Twister is based on the assumption that datasets and intermediate data can fit into the distributed memory of the computing infrastructure, which is not the case for clusters of commodity machines where each node has limited resources.

III.1.3 iMapReduce

iMapReduce [48] also proposes an implementation based on Apache Hadoop to provide a framework that can explicitly model iterative algorithms. Since every iteration performs the same operations repeatedly, iMapReduce makes use of *persistent tasks* by keeping all the tasks alive during the whole iterative process. When a persistent

task finishes processing its input, the task remains idle until the input of the following iteration becomes ready to save the resources/time needed to create, destroy, and schedule tasks for every iteration. iMapReduce divides the input of the map task into *static* data (remain unchanged during the overall iterative process) and *state* data (get updated after each iteration).

At the start of an iterative application, iMapReduce keeps a one-to-one mapping between the map and the reduce tasks and its task scheduler assigns statically every map task and its corresponding reduce to the same worker. The reduce tasks only produce state data, which are processed afterwards by the map tasks by first joining the state data with the static data and then applying the map function. This allows them to run the map tasks of the subsequent iteration asynchronously while the reduce tasks has to wait until all the map tasks of the same iteration are finished.

iMapReduce relies on persistent tasks to reduce the amount data to be shuffled from the map tasks to the reduce tasks. However, this requires that the computing infrastructure has enough resources that can accommodate all the persistent tasks simultaneously, which puts a limit over the number of concurrent MapReduce jobs that can be sharing the cluster's resources at the same time. Their static scheduling policy and coarse task granularity can lead to an unoptimized resource utilization and load unbalancing. In case of a failure, the computation is rolled back to the last saved state. iMapReduce optimizations are limited to those applications where every iteration corresponds to exactly one MapReduce jobs. And in cases where the map and reduce functions have different keys, iMapReduce has to start the map tasks of the following iteration synchronously.

Unlike the previous systems that use a static mapping between tasks and nodes, which can lead to load unbalancing, iHadoop uses a dynamic scheduling policy. The use of persistent/long-running tasks, like in Twister and iMapReduce, limits resources of the infrastructure to a certain job even if they remain idle. iHadoop does not

persist the tasks for the following iteration; since with large scale datasets, the runtime optimizes the task granularity so that the time it takes to create, destroy, and schedule tasks is insignificant compared to the time required by each task to process its input split. iHadoop is the first to our knowledge to execute asynchronously the map and reduce tasks of the following MapReduce job in an iterative process without any limitations on the supported iterative applications: Any iterative MapReduce application can run over iHadoop with the asynchronous iterations optimization.

III.2 Higher-Level Computing Frameworks Based on MapReduce

Many systems have been built on top of Hadoop, such as HadoopDB [1], Hive [40], and Pig [32]. Like Hadoop itself, they do not provide any special support for iterative applications, and are oblivious to possible optimizations for iterative computations.

Lin and Schatz [26] introduced several design patterns to efficiently process graph algorithms in MapReduce. They noticed that for many graph algorithms, the graph structures do not change from one iteration to the next. They partition graph structures into parts stored in DFS, where reducers perform remote reads to read in the static data of the computation to avoid unnecessary shuffling of invariant data from map to reduce tasks.

Pegasus [23] and Apache Mahout[28] are libraries for efficient large-scale graph mining and machine learning algorithms based on Hadoop. Most such algorithms are iterative in nature. These two systems control the iterative computation using a driver program that issues a MapReduce job for each iteration, while adjusting the corresponding input and output directories. Unlike iHadoop, they do not leverage the possible inter-iteration optimizations that can be applied to iterative algorithms.

III.3 Alternative frameworks to MapReduce

FlumeJava [5] supports data-parallel pipelines by providing a set of parallel execution primitives which are compiled into an optimized execution plan using appropriate underlying primitives (e.g., MapReduce). Pregel [29] is a distributed system designed for processing large graph datasets. Pregel's computations are a sequence of iterations that apply user defined function on each vertex in parallel. It does not support the broad class of iterative algorithms that iHadoop does.

Spark [47] is a framework built on top of a *cluster operating system* [20] allowing multiple applications/frameworks to share resources. Spark supports iterative and interactive applications while providing scalability and fault tolerance similar to those of MapReduce. Spark introduces *resilient distributed datasets* (RDD) which are read-only objects partitioned across the nodes of a cluster. RDDs can be cached in memory for a faster access and derived back from data available in reliable storage upon node failure. However, Spark does not support parallel reductions as in MapReduce which many iterative applications use to perform join operations.

CIEL [31] is a recent platform that exposes a simple programming model and transparently handles scheduling, fault tolerance, and synchronization similar to what the MapReduce model does. A CIEL job can make control-flow decisions enabling iterative and recursive computations but does not exploit the posssible asynchronous execution of iterations. Dryad [21] is a distributed framework based on a directed acyclic graph (DAG) execution model where computations are executed on *vertices*, with *edges* representing communication channels. Dryad is more expressive and flexible than MapReduce but not as straightforward to use. DryadLINQ [46] provides a high-level programming API for Dryad graphs. It supports iterative computation using standard loops. Ekanyake et al. [13] show that even with loop unrolling, DryadLINQ still has programming model overhead when performing iterative computations.

Parallel database systems are able to exploit parallelism by partitioning and replicating their storage and optimizing query execution plans accordingly [8, 36, 22]. Though they can scale linearly for tens of nodes, there are few deployments with more than one hundred nodes. Parallel databases often run on expensive homogeneous architectures and assume that failures are rare — which is not the case with large commodity clusters. We are not aware of any published parallel database deployment that was able to scale to thousands of machines.

Chapter IV

iHadoop Design

The MapReduce dataflow (as reviewed in Chapter II) enables it to scale to thousands of machines. We argue that iterative applications can take advantage of even more of the massive parallelism available in cluster environments. This chapter describes how this is accomplished in the iHadoop model, by adapting the MapReduce dataflow to the structure of iterative computations.

This chapter first examines the various dependencies and barriers present during the execution of iterative algorithms over MapReduce. Having identified the points where additional parallelism can be safely exploited, we then describe the asynchronous iterations model, followed by a description of concurrent termination checking. Introducing asynchronous iterations has implications for task scheduling, and for the model's fault tolerance, and we address both issues in this chapter.

IV.1 Dependencies in MapReduce processing

Figure IV.1 shows some of the dependencies observed in the execution of a typical iterative application in MapReduce. Only the dependencies relevant to our work are shown.

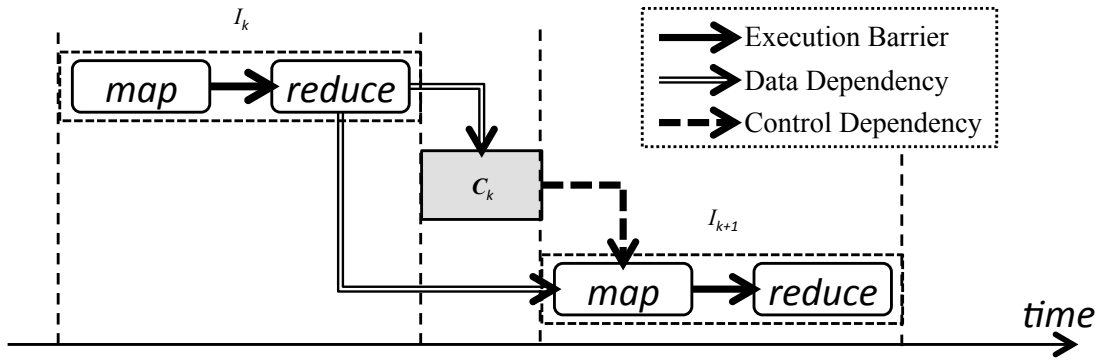


Figure IV.1: Analysis of different dependencies in the MapReduce pipeline of an iterative application.

- **Execution Barrier**: where execution of a phase has to wait until the execution of an earlier phase is *completely* finished. For example, there is an execution barrier between the Map phase and the Reduce phase in every iteration (this is true even for non-iterative MapReduce jobs), because any given reduce task must receive *all* values associated with a specific key from all mappers before it starts. This execution barrier is represented by the Shuffle phase where all reducers have to retrieve, sort, and group their respective partitions of intermediate data from mappers before they can proceed to the Reduce phase.

- **Data Dependency**: where the output of one phase is delivered as input to the next phase. For example, there is a data dependency between the Reduce phase of iteration I_k and the Map phase of iteration I_{k+1}. Similarly, when the application requires a termination check, there is a data dependency between the Reduce phase of iteration I_k and the termination check C_k. The execution barrier between Map and Reduce phases of the same iteration is a stronger type of data dependency where a reducers' input has to be fully available before starting the Reduce phase.

- **Control Dependency**: which determines if the computation should proceed

with the execution of a phase or not. For example, there is a control dependency between the termination check C_k and the Map phase of iteration I_{k+1} (or transitively the entire iteration I_{k+1}).

IV.2 Asynchronous Iterations

With typical MapReduce, every map task is associated with an input split stored in the distributed file system. Across a sequence of iterations, the input split for each map task can be

- a part of the output of the immediately preceding iteration, or
- a part of static data or the output of an earlier iteration (not the immediately preceding).

For an input split of the later case, the corresponding map task can start, with no changes in the MapReduce dataflow, asynchronously with the previous iteration since its data are already available in the distributed file system. The following discussion is concerned with those map tasks that consume parts of the output of the previous iteration as their input splits. With the typical MapReduce dataflow, those map tasks will have to wait until their input splits are available in the distributed file system, i.e., until the previous iteration is completely finished.

While shuffling is necessary between the Map and Reduce phases of the same iteration I_k, it is not required on the output of iteration I_k before it is handed to the Map phase of the next iteration I_{k+1}. Therefore, a mapper of I_{k+1} can generally process any part of the input data in any order. The same is true for mappers in traditional (non-iterative) MapReduce jobs.

This means there is *no* execution barrier between the Reduce phase of I_k and the Map phase of I_{k+1} despite the data dependency; there is nothing in theory that

prevents mappers of I_{k+1} from starting execution whenever input data are available, even while the reducers of I_{k+1} are still processing their input and emitting their output. Therefore, we propose that the output of I_k can be fed as input to the mappers of I_{k+1} as long as the reducers of I_k are emitting output. Figure IV.2 shows how the data dependency between the Reduce phase of iteration I_i and the Map phase of iteration I_{k+1} can be translated into a communication channel.

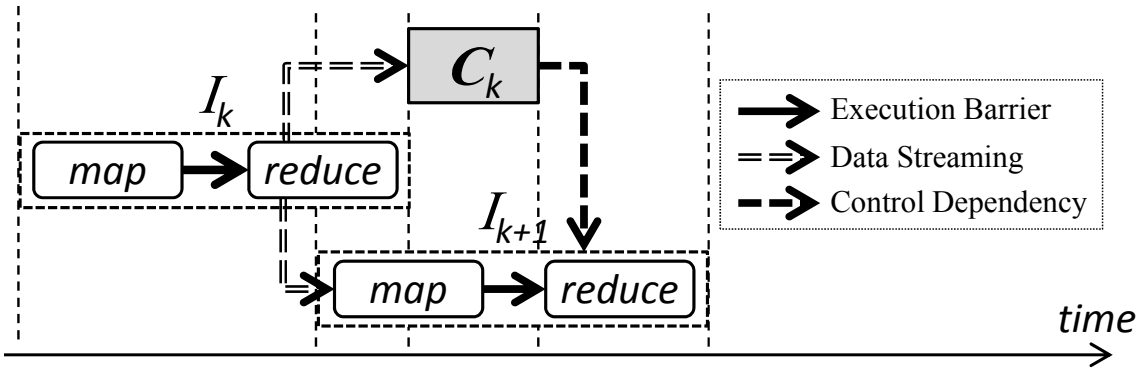


Figure IV.2: Asynchronous pipeline

With asynchronous iterations, every reducer opens two streams: (1) a stream to the underlying file system, similar to MapReduce, to maintain the same fault tolerance model of MapReduce, and (2) a socket stream to mappers of a following iterations, used for direct transfer of the inter-iteration data.

The performance gain expected by asynchronous iterations is not due to reduction in (or better efficiency of) the needed computations, but rather due to the changes in dataflow that allow iterations to exploit more parallelism. Consider two successive iterations I_k and I_{k+1}. Running an iterative MapReduce job in the traditional *synchronous* fashion will allocate as much as possible of the resources available to the job to I_k until it is completed. Any other available resources that cannot be utilized by the current running iteration will remain idle. Using *asynchronous* iterations, some of the available job resources are allocated to I_{k+1} whenever I_{k+1} can start processing

the input data.

IV.3 Concurrent Termination Checking

Two methods are commonly used in determining when an iterative computation should stop. The first is to specify beforehand the number of iterations that should be executed. Alternatively, the computation can check a *termination condition* at the end of each iteration to decide whether to continue the computation or to stop. A typical termination condition compares the output of the last iteration with that of its preceding one, and terminates the computation if the difference does not exceed a specified threshold. A termination check may, however, be more sophisticated: for example, it may involve reading the outputs of more than one previous iteration.

In the context of iterative algorithms in which iterations are implemented as MapReduce jobs, the termination condition itself may be implemented as a separate MapReduce job, or it can be an arbitrary script.

Figure IV.1 shows that a termination check at the end of iteration k, C_k, is involved in two types of dependencies:

- *Data dependency* with the Reduce phase of the previous iteration I_k; this data dependency can be handled in a way similar to the data dependency between that Reduce phase of I_k and the Map phase of I_{k+1}.

- *Control dependency* with iteration I_{k+1}.

If the iterative application will eventually run for n iterations, the termination condition will not be satisfied for the the first $n - 1$ iterations, and will only be satisfied at the last one. iHadoop speculates that the termination condition will not be satisfied for *every* iteration, and thus the next iteration can immediately start without delay. In the asynchronous setup, described earlier and shown in Figure IV.2,

iteration I_{k+1} ignores the control dependency and starts even while iteration I_k is running. Whenever C_k is checked and found satisfied (i.e., iterative computation has to stop), the iHadoop framework sends a termination signal to I_{k+1} cleaning up all the tasks belonging to it.

iHadoop runs the termination check in the background concurrently with asynchronous iterations. For $n-1$ out of n iterations, iHadoop takes advantage of the correct speculation that allows the next iteration to make significant progress upon the completion of the earlier iteration. Although the following iteration will be useless and a waste of resources if the condition is satisfied, it is insignificant compared to the time savings asynchronous iterations can achieve. Chapter V discusses when precisely the termination condition gets checked.

IV.4 Task Scheduling

Every reduce task of iteration I_k feeds its output to one or more map tasks of iteration I_{k+1}. The iHadoop task scheduler takes advantage of this inter-iteration locality by scheduling tasks that have a direct data dependency on the same physical machine. Map tasks of the first iteration are scheduled using MapReduce's default task scheduler. For subsequent iterations, the iHadoop task scheduler tries to schedule a reduce task and its associated set of map tasks (those tasks that exhibit a producer/consumer relation) on the same physical machine.

While the MapReduce task scheduler tries to schedule map tasks on a machine that has a local replica of its assigned input split, the iHadoop task scheduler achieves the same goal since a map task will receive its input split streamed from a reduce task running on the same physical machine. With large clusters, network bandwidth is a relatively scarce resource and the iHadoop task scheduler strives to reduce the network traffic and hence the overall execution time. If iteration I_{k+1} has parts of

its inputs that are not produced from iteration I_k (like static data in the distributed file system), iHadoop task scheduler schedules the map tasks that will be processing those parts of the input according to the default MapReduce policy.

MapReduce implementations have traditionally considered where computations (tasks) should be executed (i.e., on which node). Since iHadoop supports asynchronous iterations, it raises a new factor that should be taken into consideration by implementations, namely *when* to schedule tasks for optimal execution. We discuss this in more detail, in the context of our implementation, in Section V.1.

IV.5 Fault Tolerance

The commodity cluster nodes that are used in most MapReduce deployments contain multiple hard disks, memory subsystems, processors, network cards, and other components that are prone to failures. In large scale systems with many nodes, failures should be treated as the normal case rather than the exception [43], and thus MapReduce was designed to tolerate machine failures gracefully.

MapReduce handles machine failures transparently by re-executing all completed and in-progress map tasks as well as in-progress reduce tasks which were allocated on the failed machine. iHadoop builds on top of this fault tolerance mechanism; it handles the following two additional failure scenarios in the context of the asynchronous iterations:

- failures of in-progress reduce tasks that are streaming their outputs to one or more map tasks of the immediately following iteration, and
- failures of map tasks that are consuming their input directly from a reducer of the immediately preceding iteration.

Upon failure of an in-progress reduce task, iHadoop re-starts in-progress map tasks that are receiving input from the failed reducer, whereas completed map tasks

do not have to be re-started. Since the reduce operator is deterministic, we only need a deterministic reduce-to-map partitioning scheme to ensure that each map task of the immediately subsequent iteration will process the same input in case of a reduce task restart.

Every reduce task in asynchronous iterations sends its output to the set of linked mappers of the following iteration and to the distributed file system. Upon failure of a map task that is consuming its input directly from a reducer that did not fail, iHadoop restarts the map task but its input is retrieved in this case from the distributed file system if it is ready. Otherwise, the map task waits until the input is ready in the distributed file system.

IV.6 Load Balancing

The MapReduce model contributes its load balancing behaviour to the following two design issues.

1. Task granularity: The MapReduce divides the whole job into a number of map and reduce tasks that far exceeds the number of worker nodes in the cluster. This improves the dynamic load balancing by being able to allocate an approximate equal share of the work to each worker node.

2. Dynamic scheduling: The scheduling decisions of MapReduce are based on the information passed to the master node through the heartbeats of the worker nodes. Each worker updates the master with its current load and available resources periodically. The master uses this information to schedule tasks dynamically wherever there are available resources.

Similarly, iHadoop load balancing inherits its behaviour from MapReduce. iHadoop does not require any changes to the task granularity. However, the iHadoop task

scheduler tries to schedule every pair of tasks that exhibit a producer/consumer relation together on the same physical machine. Ideally, every reduce task of iteration I_k will have an equal share of the work and will be producing an equal share of the output, which will result in an equal share of work to the map tasks of iteration I_{k+1}. As an extension to improve the load balancing for an iterative application over iHadoop, a map task of iteration I_{k+1} can be scheduled to run on any less loaded worker node. But in the later case, its input will be streamed over the network rather than the fast local inter-process transfer.

IV.7 Summary

This chapter analyzed the several dependencies that exist in an iterative MapReduce job. For an iteration I_{k+1} that is preceded by the "in-progress" iteration I_k, its input can be (1) the output of I_k—which is not yet available in DFS (data dependency between I_k and I_{k+1}), (2) some data *already* available in DFS, or (3) a combination of the two. Those map tasks of I_{k+1} that will be processing data already available in DFS can start asynchronously with I_k with no changes in the default MapReduce dataflow. For those map tasks of I_{k+1} that will be processing the output of I_k, they, as well, can start asynchronously by processing the output of the reduce tasks of I_k as it becomes available. Those map tasks do not have to wait for the reducers of I_k to write their output to DFS: the reducers of I_k send their outputs directly to the map tasks of I_{k+1}. The data dependency can be converted into a communication channel between the reducers of I_k and the mappers of I_{k+1}.

As for the control dependency, which exists between the termination check C_k at the end of iteration k and iteration I_{k+1}, this control dependency can be ignored speculating that the termination condition will not be satisfied. For an iterative job that will be running for n iterations before its termination condition can be met, this

speculation is correct for $(n-1)$ times which further reduces the application's latency.

The task scheduler of iHadoop tries to maximize data locality. It schedules tasks that have a direct data dependency on the same physical machine which provides a faster data transfer and reduces the network traffic which is a scarce resource in a MapReduce cluster. The dataflow changes introduced by iHadoop do not threaten the strong fault tolerance model of MapReduce since reducers' outputs are still being written to DFS. The load balancing of iHadoop also draws from MapReduce's load balancing since iHadoop still maintains a dynamic task scheduling.

Chapter V

iHadoop Implementation

In this chapter, we present iHadoop, our implementation of the iHadoop model. As discussed in Section II.4, the MapReduce framework is unaware of the behaviour of an iterative application. In order for iHadoop to be able to apply the optimizations presented in Chapter IV, it has to be aware of the iterative nature of an application. Figure V.1 shows how the iHadoop framework controls the iterative execution of an application.

Comparing Figure II.2 and Figure V.1 illustrates the different application-framework interaction of the MapReduce model and the iHadoop model, respectively. With iHadoop, the program submits an iterative job to the framework which, in turn, will be responsible for running every iteration and the termination checks if required. With MapReduce, the driver program has to break down and submit each iteration as a series of MapReduce jobs, adjusting input/output directories for every job, and running the termination check after each iteration if needed.

iHadoop is built on top of HaLoop [4] and Hadoop [18]. HaLoop is an extension of Hadoop, providing programming support for iterative applications where the framework is aware of their behaviour. HaLoop provides other optimizations for iterative MapReduce applications as discussed earlier in III.1. Please refer to Appendix A for

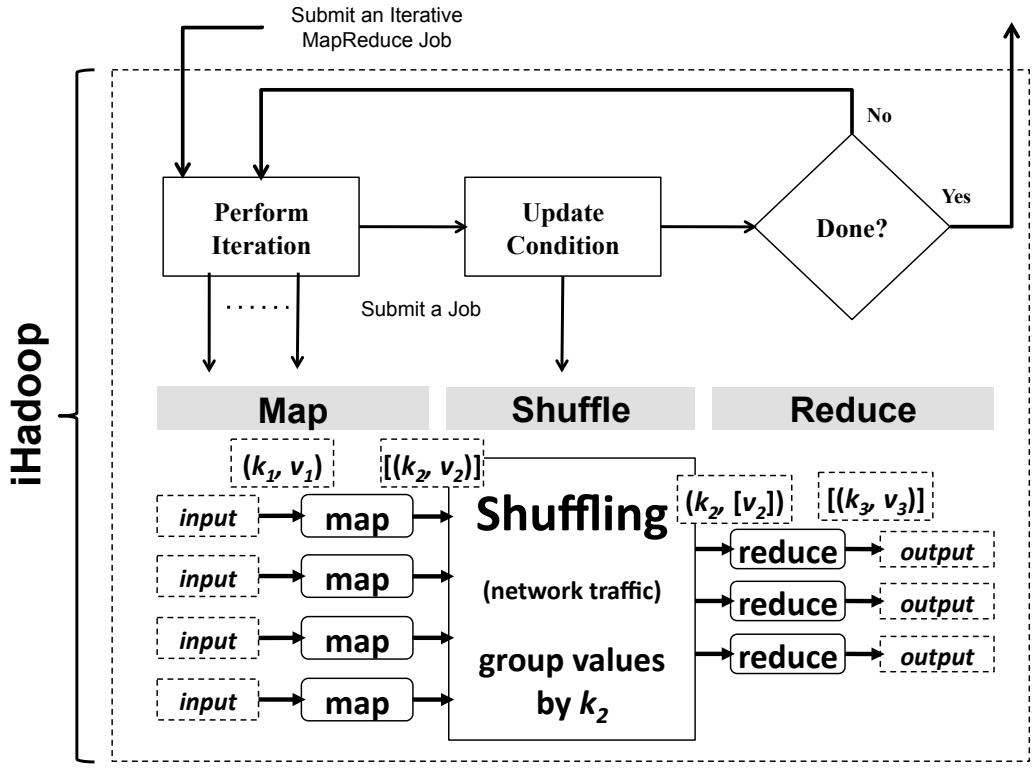


Figure V.1: The iHadoop framework is responsible for running every iteration and the termination check between each.

details about the programming model of HaLoop and other API extensions proposed by iHadoop. The optimizations proposed by iHadoop are orthogonal to the features of HaLoop (such as caching invariant data). As our evaluation will show in Chapter VI, the optimizations can be used side by side to boost performance. This chapter discusses some optimizations and issues encountered when implementing iHadoop:

- Reducer early start: An iteration that is running asynchronously with an earlier iteration achieves significant Map progress. Also, iHadoop can achieve a greater progress by starting the reducers of the asynchronous iteration early to work on the Shuffle phase.

- Reducer-to-mappers streaming: Increasing the number of I_{k+1} mappers that receive their input data from an I_k reducer increases the overhead of creating the communication channels. However, if this number is small, the Shuffle phase

of I_{k+1} will progress slower.

- Mapper JVM reuse: To minimize the overhead of creating the communication channels between I_k reducers and I_{k+1} mappers, iHadoop persists the communication channel for every reducer.

The following discusses the previous optimizations and other implementation issues in more detail.

V.1 Reducer Early Start

Since iHadoop runs iterations asynchronously, the task scheduler is optimized to launch tasks in a manner which achieves better utilization of resources. We denote a reducer from iteration I_k that is in the phase P by $R_{k,P}$. In our asynchronous setting, a reducer $R_{k,Reduce}$ applying the *reduce* function to its input will be streaming its output to mappers of iteration I_{k+1}. In this scenario, a reducer $R_{k+1,Shuffle}$ of iteration I_{k+1} can start collecting the intermediate outputs of the mappers of iteration I_{k+1} as they finish. We call this optimization *reducer early start* where reducers of two consecutive iterations I_k and I_{k+1} are processing and shuffling, respectively, their input at the same time.

Figure V.2 shows this optimization in action. As reducers of iteration I_9 are streaming their outputs to mappers of iteration I_{10}, the reducer early start will try to schedule reducers of I_{10} to start collecting intermediate data from the mappers of I_{10} as they finish. Note that by the time I_9 finishes, the mappers of I_{10} have made significant progress and reducers of I_{10} were able to collect much of the intermediate data.

A reducer starts the Shuffle phase by copying the intermediate output corresponding to its partition of the key space from all the map tasks. For reducers of iteration

```
..
..
00:33:16 INFO JobClient:  I:9 ( map 100% reduce 32%) I:10 ( map 0% reduce 0%)
00:33:37 INFO JobClient:  I:9 ( map 100% reduce 46%) I:10 ( map 0% reduce 0%)
00:33:39 INFO JobClient:  I:9 ( map 100% reduce 47%) I:10 ( map 2% reduce 0%)
00:33:42 INFO JobClient:  I:9 ( map 100% reduce 55%) I:10 ( map 2% reduce 0%)
00:33:43 INFO JobClient:  I:9 ( map 100% reduce 56%) I:10 ( map 4% reduce 0%)
00:33:45 INFO JobClient:  I:9 ( map 100% reduce 63%) I:10 ( map 7% reduce 0%)
00:33:46 INFO JobClient:  I:9 ( map 100% reduce 64%) I:10 ( map 10% reduce 0%)
00:33:52 INFO JobClient:  I:9 ( map 100% reduce 75%) I:10 ( map 20% reduce 1%)
00:33:55 INFO JobClient:  I:9 ( map 100% reduce 77%) I:10 ( map 27% reduce 3%)
00:34:02 INFO JobClient:  I:9 ( map 100% reduce 82%) I:10 ( map 42% reduce 7%)
00:34:09 INFO JobClient:  I:9 ( map 100% reduce 86%) I:10 ( map 55% reduce 11%)
00:34:20 INFO JobClient:  I:9 ( map 100% reduce 91%) I:10 ( map 74% reduce 18%)
00:34:41 INFO JobClient:  I:9 ( map 100% reduce 97%) I:10 ( map 93% reduce 29%)
00:34:59 INFO JobClient:  I:9 ( map 100% reduce 99%) I:10 ( map 99% reduce 31%)
00:35:01 INFO JobClient:  I:9 ( map 100% reduce 100%) I:10 ( map 100% reduce 32%)
00:35:02 INFO JobClient:  I:10 ( map 100% reduce 32%) I:11 ( map 0% reduce 0%)
00:35:08 INFO JobClient:  I:10 ( map 100% reduce 53%) I:11 ( map 0% reduce 0%)
00:35:09 INFO JobClient:  I:10 ( map 100% reduce 60%) I:11 ( map 0% reduce 0%)
00:35:15 INFO JobClient:  I:10 ( map 100% reduce 67%) I:11 ( map 1% reduce 0%)
00:35:48 INFO JobClient:  I:10 ( map 100% reduce 72%) I:11 ( map 14% reduce 0%)
00:36:11 INFO JobClient:  I:10 ( map 100% reduce 74%) I:11 ( map 22% reduce 1%)
00:36:46 INFO JobClient:  I:10 ( map 100% reduce 78%) I:11 ( map 34% reduce 5%)
..
```

Figure V.2: A sample iHadoop execution log of asynchronous iterations.

I_{k+1} to be able to shuffle the outputs of the mappers of the same iteration efficiently, each reducer of iteration I_k should be streaming its output to several map tasks of iteration I_{k+1}. We investigate this in more detail in the next section.

V.2 Reducer to Mappers Streaming

Figure V.3 illustrates how streaming is implemented in iHadoop. The left hand side shows "1:1 streaming" where a reducer of I_k is streaming all its emitted output to a single mapper of I_{k+1}. Since there is an execution barrier between the Map and Reduce phases within a single iteration, the reducer of I_{k+1} has to wait until its mapper finishes before the framework starts to shuffle the intermediate output[1].

[1]A reducer receives its input by shuffling the intermediate output of multiple mappers based on the key sub-space assigned to it. Here, we focus on the case of one mapper.

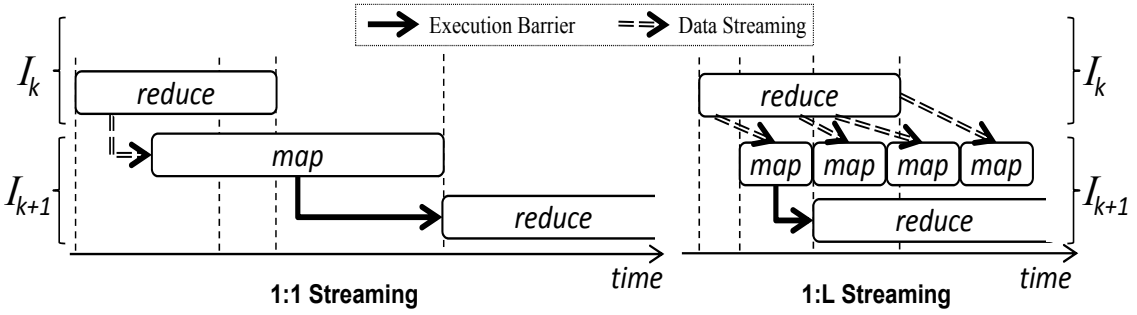


Figure V.3: Reducer to mappers streaming. "1:1 streaming" (left) requires one-to-one mapping between reducers of iteration I_k and mappers of iteration I_{k+1}.

To mitigate the potentially long delay, iHadoop implements a "1:L streaming" mechanism where the reducer streams its intermediate output to L *linked* mappers of I_{k+1} in succession, each of which will receive an equal portion of that output and thus can potentially finish in L times less than the case for "1:1 streaming". This allows a reducer of I_{k+1} to start collecting its input earlier. Notice that the reducer of I_k still streams to one mapper at a time. The number of linked mappers, L, has to be chosen carefully so that the overhead of creating that number of mappers and new connection streams does not degrade overall performance; we investigate the effect of varying L in Subsection VI.3.4.

V.3 Mapper JVM Reuse

To decrease the overhead of creating new connection streams between a reducer and the corresponding linked mappers of the following iteration, iHadoop reuses the same Java Virtual Machine (JVM) for these mappers. Before a reducer of iteration I_k starts the Reduce phase, iHadoop's `TaskScheduler` ensures that there is a mapper of iteration I_{k+1} running on the same physical machine waiting to establish a socket connection with that reducer. Once a connection is established, the JVM will manipulate that connection as a shared object between subsequent map tasks. When

the map task finishes, the JVM promptly requests the next map task to run from the `TaskTracker` which will use the shared connection stream to read its input from the reducer.

After applying the *map* function on their inputs, mappers need to create partitions for every reducer and commit them to disk. These operations can take a long time and delay the flow of data from a reducer to the next linked mapper. To speed up this transition, every map task, after applying the *map* function, forks two threads: the first is a low priority thread that commits the intermediate output to local disk, and the other of higher priority is to runs the next map task.

V.4 Concurrent Termination Check

A termination check C_k typically runs after the conclusion of each iteration I_k. If this check is implemented as a MapReduce job, then a similar streaming technique (to the one discussed earlier from reducers of I_k to mappers of I_{k+1}) can be adopted. This enables concurrent execution of an iteration and its termination check.

C_k can generally be any user-defined program (i.e., not necessarily a MapReduce job), so the current implementation of iHadoop only launches the termination check C_k after iteration I_k commits its output to HDFS. In Subsection VI.3.2 we measure the performance gains achieved from running asynchronous iterations while the termination check executes concurrently.

V.5 Ensuring Unique Task Identifiers

Each task in Hadoop has a unique identifier (Task ID). This identifier is the concatenation of `JobTracker_ID`, `Job_ID`, `(m | r)`, and `Partition_ID`. For example, the following string: `task_201105010250_0001_m_000002` represents a map task with ID `000002` from a job with ID `0001`. If we use the same convention in iHadoop,

this identifier will no longer be unique. Consider task A of iteration I_k and task B of iteration I_{k+1} running at the same time and both are representing the map task with ID `000002`; both would have the identifier `task_201105010250_0001_m_000002`. To avoid this problem, we add the iteration number to the task identifier. iHadoop would use the identifier `task_201105010250_0001_k_m_000002` to guarantee unique identifiers across different iterations.

Chapter VI

Experimental Evaluation

The design and implementation of iHadoop raise many questions that an experimental evaluation is able to answer, most notably: How significant is the performance improvement due to asynchronous execution of iterations? How much is to be gained by checking the termination condition concurrently with the subsequent iteration? Does the number of linked mappers affect the performance? This chapter evaluates the performance of `iHadoop` and discusses the results obtained through the experiments.

VI.1 Evaluated Systems

System	Iterative API	Asynchronous	Caching	Note
`Hadoop`	-	-	-	Baseline
`iHadoop`$_a$	✓	✓	-	
`iHadoop`$_c$	✓	-	✓	≡ HaLoop
`iHadoop`$_{ac}$	✓	✓	✓	

Table VI.1: Systems compared in the experiments.

This work evaluates iHadoop and compares it to other variants that have different optimizations enabled. The implementation of `iHadoop` is based on Hadoop 0.20.2 which is the baseline to all the evaluated systems. `Hadoop`, the baseline, uses a driver program that controls the execution of a sequence of iterations, each is a

set of MapReduce jobs. If a condition needs to be satisfied in `Hadoop` before termination, this check is executed as a separate MapReduce job that is launched after the completion of each iteration. The x in `iHadoop`$_x$ indicates features denoted by single letters: 'a' for 'asynchronous support' and 'c' for 'caching invariant data' as listed in Table VI.1. `iHadoop`$_a$ includes the optimizations discussed in this work, i.e., asynchronous iterations and concurrent termination check. `iHadoop`$_c$ caches invariant data between iterations. We used HaLoop to obtain results for this optimization. However, we keep the name '`iHadoop`$_c$' for the sake of consistency. `iHadoop`$_{ac}$ is our implementation of the asynchronous iterations support while caching and indexing invariant data between iterations based on the HaLoop caching technique.

VI.2 Experimental Setup

We ran the experiments on the two clusters described in Table VI.2. Each cluster has a separate `NameNode`/`JobTracker` machine with 2.67 GHz Intel® Xeon® X5550 processor and 24GB of RAM. We used the two datasets described in Table VI.3: LiveJournal[1], and WebGraph[2]. All the results presented in this section are obtained without any node/task failures and no other user processes/jobs were concurrently running with our jobs on the clusters. By default, we set the number of reducers R to the number of the nodes in the cluster where we run the experiment. A maximum of 6 map tasks and 2 reduce tasks were configured per node. Unless otherwise stated, the output of each iteration is written to HDFS with replication factor of 3 upon its completion, the number of iterations is pre-specified with no termination check, and `iHadoop`$_a$ and `iHadoop`$_{ac}$ have the number of linked mappers L set to 5.

Three iterative applications were used in the evaluation: PageRank (PR), Descendant Query (DQ), and Parallel Breadth-First Search (PBFS). These applications

[1]http://snap.stanford.edu/data/soc-LiveJournal1.html
[2]http://lemurproject.org/clueweb09.php/

	$Cluster_{12}$	$Cluster_6$
Nodes	12	6
Cores/Node	4	
Threads/Core	2	
Total Cores	48	24
RAM/Node	4GB	
Processor	Intel ® Core™ i7-2600	
CPU Speed	3.40 GHz	
Network	1Gbit	

Table VI.2: Cluster configurations.

Dataset	Nodes	Edges	Size	Description
LiveJournal	4,847,571	68,993,773	1GB	Semi-synthetic social network graph
WebGraph	50,013,241	151,173,117	2.5GB	Graph of 1^{st} English segment of ClueWeb09

Table VI.3: Datasets used in the experiments.

represent two different classes of iterative algorithms; PR and PBFS represent those algorithms where the input of an iteration is mainly the output of the preceding one, while DQ represents those where the input of an iteration depends on previous iterations along with the immediately preceding one. PR is a link analysis algorithm. Each iteration is represented as (i) a join operation between ranks and the linkage structure, followed by (ii) a rank aggregation step. The input for the join step is the output of the previous aggregation step and some (cachable) invariant data (e.g., linkage structure). The input for the aggregation step is the output of the previous join step only. PBFS is a graph search algorithm. Each iteration is represented by a single MapReduce job. The mappers of this job read the output of the earlier iteration, update the distance of each node, and emit the updated records to the reducers. For each node, the reducers set the smallest distance discovered so far. DQ is a social networking algorithm, each iteration is represented as (i) a join operation between vertices discovered from the previous iteration and the linkage structure, followed by (ii) a duplicate elimination step. The input for the join step is the output of the

previous duplicate elimination step and some invariant metadata. The input for the duplicate elimination step is the output of the previous join step and the outputs of *all* previous duplicate elimination steps. For more details about these applications, please refer to Appendix B.

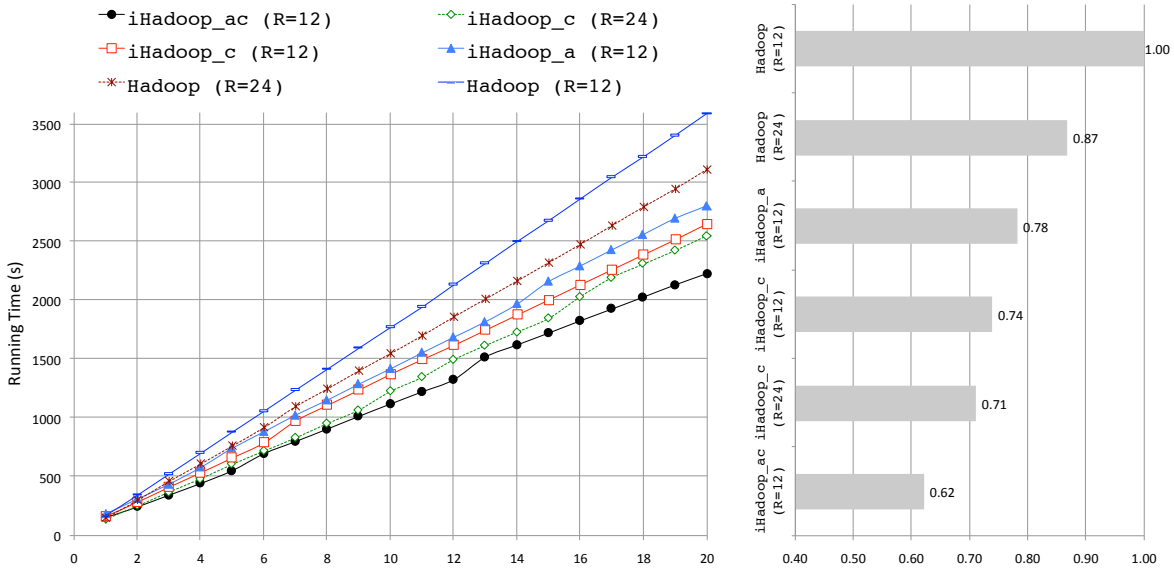
VI.3 Results

In this section, we evaluate and break down the effect of asynchronous iterations and the concurrent termination check on the execution time of iterative applications. For the systems that use asynchronous iterations (i.e., $\texttt{iHadoop}_a$ and $\texttt{iHadoop}_{ac}$), we use R reducers for each iteration, which allows another R reducers to be used concurrently for the subsequent iteration. For a fair comparison, we run the systems that do not use asynchronous iterations (i.e., `Hadoop` and $\texttt{iHadoop}_c$) with two settings: R and $2R$ reducers. However, the performance improvement for those runs with double the number of reducers is not as significant as those achieved via the other optimizations. For the sake of simplicity, the relative performance numbers reported explicitly in this section are those comparing runs of equal number of reducers, while figures show the additional data points for the different number of reducers. The results of the experiments were compared to the results of `Hadoop` to check the correctness, the only differences found were due to the floating-point operations.

VI.3.1 Asynchronous Iterations Evaluation

PageRank. Figure VI.1 shows the performance of running PR on Cluster_{12} for 20 iterations using the WebGraph dataset; Figure VI.1(a) represents the overall running time of the entire job, each iteration is comprised of both the join and aggregation steps of PR. Figure VI.1(b) illustrates the average execution time per iteration, normalized to the baseline (`Hadoop`).

The results indicate that iHadoop$_a$ is significantly faster than Hadoop, reducing the average running time to 78%; a speedup of 1.28 that is due to using asynchronous iterations. iHadoop$_c$ adopting the caching of invariant data, is able to reduce the running time to 74%. The combination of asynchronous iterations and caching, featured in iHadoop$_{ac}$, exhibits the best performance, 62% of the baseline execution time, a speedup of 1.61 over Hadoop. The asynchronous behavior of iHadoop$_{ac}$ reduces total running time to 84% when compared to iHadoop$_c$.



(a) Total running time of 20 iterations

(b) Normalized average iteration time

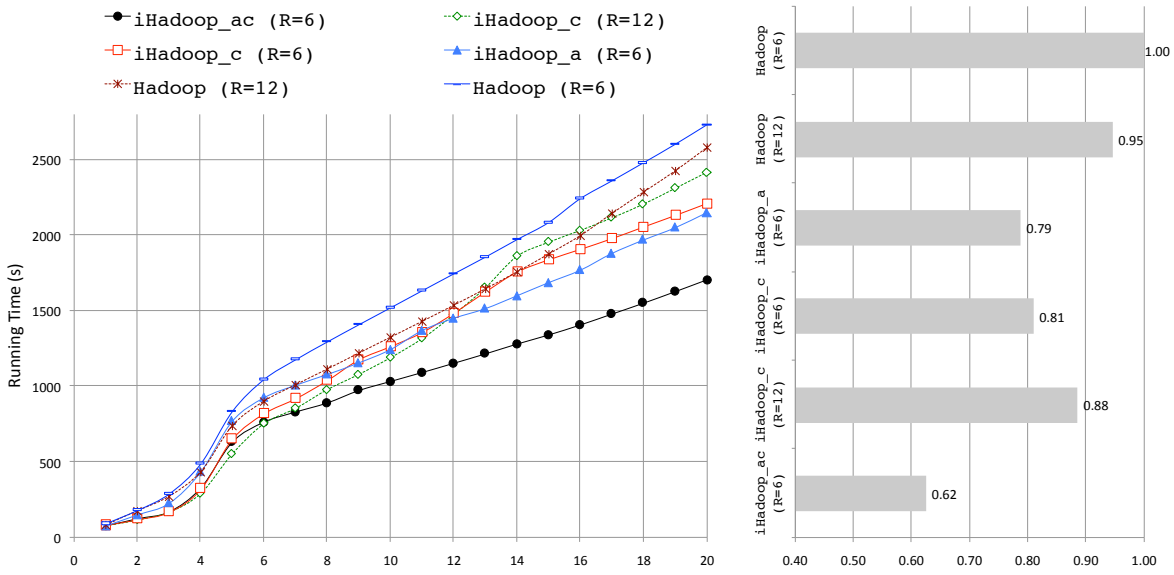
Figure VI.1: Performance of PageRank (WebGraph Dataset - Cluster$_{12}$)

Descendant Query. Figure VI.2 shows the performance of DQ on Cluster$_6$ for 20 iterations using the LiveJournal dataset. iHadoop$_a$ and iHadoop$_{ac}$ had L set to 4 in this experiment since the amount of output that each reducer produces in DQ are relatively smaller. iHadoop$_a$ reduces the total running time to 79% compared with Hadoop. iHadoop$_c$ achieves a similar improvement of 81% over Hadoop. Moreover, iHadoop$_{ac}$ reduces the total run time to 62% of the baseline.

In DQ, as the number of iterations increases, the input size for one iteration is

dominated by the outputs of *earlier* previous iterations rather than the immediately proceeding iteration.

For iteration I_{k+1} in $\texttt{iHadoop}_a$ and $\texttt{iHadoop}_{ac}$, the ratio of data that is being transferred via local streaming (the output of iteration I_k) compared to the data that is being transferred from HDFS (the outputs of iterations I_{k-1}, I_{k-2}, ..) is getting smaller as the iteration number increases. This mitigates the effect of the fast local data transfer. This demonstrates that starting an iteration earlier has a larger effect on the time savings achieved by $\texttt{iHadoop}_a$ and $\texttt{iHadoop}_{ac}$ compared to the fast local data transfer between a reducer and its linked mappers.



(a) Total running time of 20 iterations

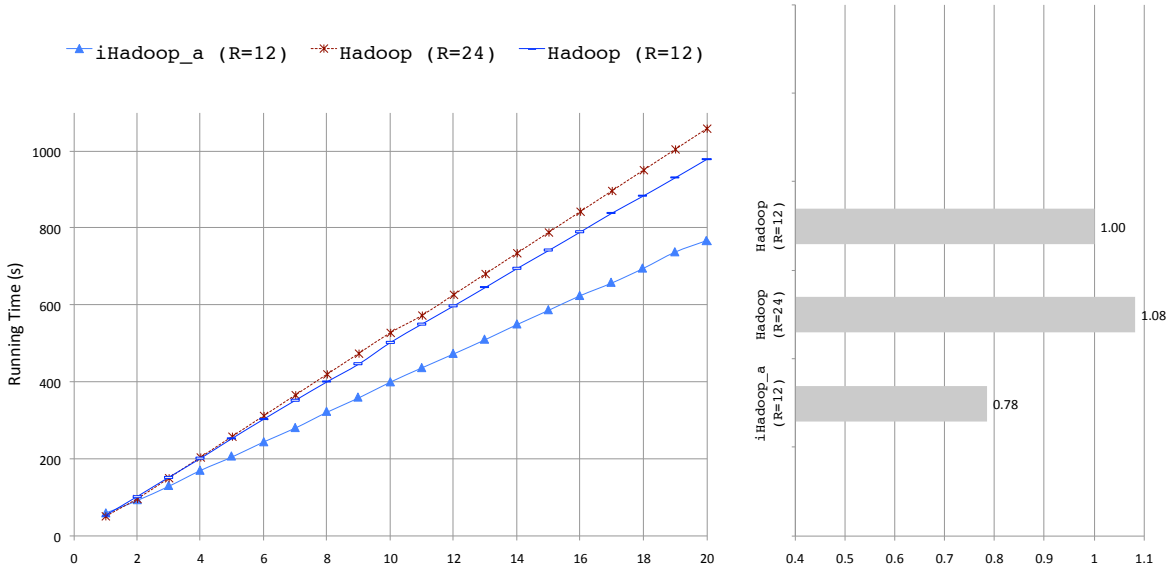
(b) Normalized average iteration time

Figure VI.2: Performance of Descendant Query (LiveJournal Dataset - Cluster_6)

Parallel Breadth-First Search. Figure VI.3 shows the performance of PBFS on Cluster_{12} for 20 iterations using the WebGraph dataset[3]. $\texttt{iHadoop}_a$ reduces the total running time to 78% compared with `Hadoop`. Figure VI.3 is missing the values for $\texttt{iHadoop}_c$ and $\texttt{iHadoop}_{ac}$, since we cannot apply the caching techniques to a "one-

[3]The WebGraph dataset was converted into adjacency list format for this application. The new format was 2.1GB.

step" PBFS since each iteration reads the whole graph and generates an updated version of the graph as a whole.



(a) Total running time of 20 iterations

(b) Normalized average iteration time

Figure VI.3: Performance of Parallel Breadth-First Search (WebGraph Dataset - Cluster_{12})

Overall, the results for these applications show clearly that asynchronous iterations significantly improve the efficiency of iterative computations, under different configurations that use a variety of optimization settings.

VI.3.2 Concurrent Termination Check

The goal of the next experiment is to measure the performance gain that `iHadoop` achieves when termination checking is performed concurrently with the subsequent iteration. We ran PR on Cluster_6 using the LiveJournal dataset with a user-defined termination condition that was satisfied at the end of the ninth iteration. Figure VI.4 compares the performance of `Hadoop` and $\texttt{iHadoop}_a$. The shaded blocks indicate the execution time of the iterations while the white ones indicate the execution time of the termination check. Since the check is always performed concurrently with the

iterations, it was never featured in $\texttt{iHadoop}_a$. As a side effect of the concurrent execution, a portion of the tenth iteration was already performed when the condition was finally satisfied. We call this portion *wasted computation*. The black block in Figure VI.4 represents the time $\texttt{iHadoop}_a$ spent after the ninth iteration in (a) the termination check and (b) wasted computation. We quantify the wasted computation by subtracting the average termination check time from the total time spent after the ninth iteration. However, the wasted computation is less than 5% of the total running time.

Running the termination check concurrently with the asynchronous iterations of $\texttt{iHadoop}_a$ introduced an overhead of 7% as measured by comparing this execution to the execution of $\texttt{iHadoop}_a$ without a termination check. Whereas the termination check for `Hadoop` introduced 24% overhead to the running time.

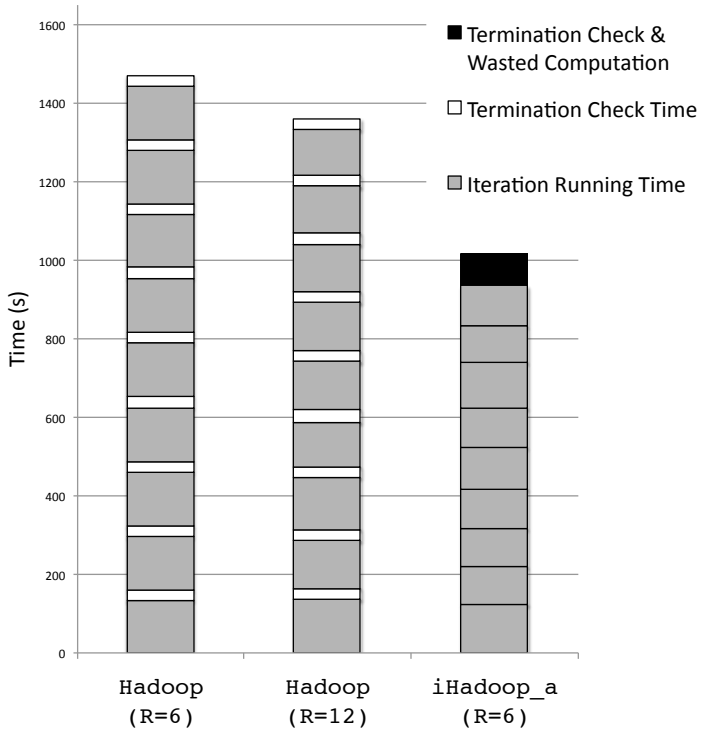


Figure VI.4: Concurrent vs. synchronous termination condition check (PageRank - LiveJournal - Cluster_6)

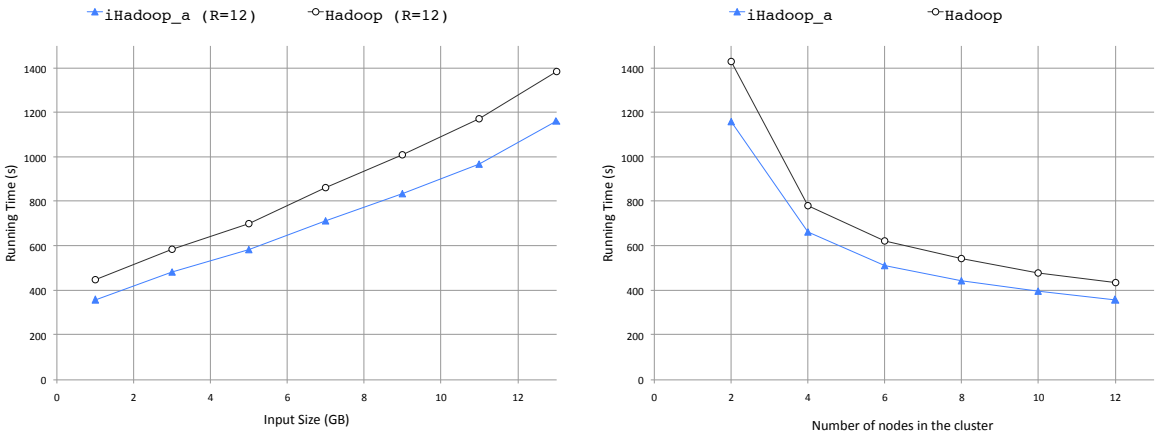
Overall, the experiment shows that asynchronous iterations and concurrent termination checking in iHadoop$_a$ reduce the overall execution time to 69%, achieving an average speed up of 1.45 per iteration over Hadoop.

VI.3.3 iHadoop Scalability Tests

In this subsection, we examine the scalability of the asynchronous behaviour of iHadoop$_a$. We used PageRank over the LiveJournal dataset for 5 iterations in the following set of experiments. In Figure VI.5 (a), we compare the performance of iHadoop$_a$ and Hadoop when varying the input size. We replaced the identifiers in LiveJournal with longer strings to generate several sizes of the dataset.

In Figure VI.5 (b), we compare the performance of iHadoop$_a$ and Hadoop when varying the number of nodes in the cluster. For both, iHadoop$_a$ and Hadoop, we set the number of reducers to the number of nodes.

Overall, iHadoop experiences the same scalability behaviour as Hadoop.



(a) Performance vs. different input sizes (b) Performance vs. different cluster sizes

Figure VI.5: Performance of iHadoop$_a$ and Hadoop when varying the input size and the number of nodes

VI.3.4 iHadoop Performance Tuning

The following set of experiments show the effect of some parameters over the performance of iHadoop. The first experiment shows the effect of varying the number of linked mappers for each reducers. Next, we show how much gain achieved through the local streaming between each reducer and the mappers of the following iteration. Lastly, we show the effect of skipping writing the output of reducers to the distributed file system.

Figure VI.6: Number of Linked Mappers Per Reducer vs. Performance (PageRank - LiveJournal - Cluster_6)

Number of Linked Mapper. As discussed in Section V.2, the number of linked map tasks per reducer L can be critical to iHadoop performance. While increasing L might increase the overhead of creating multiple mappers, it might also help reducers (of the same iteration of the linked mappers) collect intermediate data faster as mappers finish. Figure VI.6 illustrates the effect of varying L on the total execution time of 5 iterations of PR using the LiveJournal dataset on Cluster_6. From the figure, $\texttt{iHadoop}_{ac}$ performs best at $L = 5$ for PageRank with the LiveJournal dataset. The

optimal value for L depends on the amount of data every reducer generates.

Network vs. Local Streaming. The task scheduler of iHadoop exploits inter-iteration locality by scheduling tasks that exhibit a producer/consumer relationship on the same physical machine. This local data transfer is faster and saves bandwidth. The next experiment tries to quantify the gains achieved from such scheduling. In Figure VI.7, we compare, using PageRank on the LiveJournal dataset, the performance of `iHadoop`$_a$ when every reduce task and its set of linked mappers are scheduled on the same physical machine (local) versus the performance of `iHadoop`$_a$ when forcing every reduce task to run on a machine other than those machines running its set of linked map tasks (network). In `iHadoop`$_a$ (network), every reduce task sends its output to its linked map tasks over the network. Overall, `iHadoop`$_a$ (local) was 9% faster due to the faster data transfer between reducers and their linked mappers and the overall savings in bandwidth.

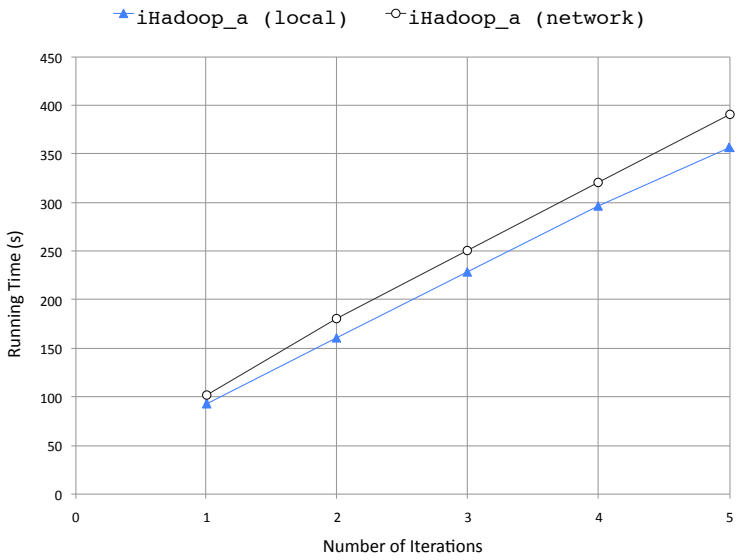


Figure VI.7: Performance of `iHadoop`$_a$ when reducer-mappers streaming happens locally vs. over the network (PageRank - LiveJournal - Cluster$_{12}$)

Skipping HDFS. In all the experiments presented so far, `iHadoop`$_a$ and `iHadoop`$_{ac}$

write their outputs to HDFS in every iteration concurrently with streaming it to the mappers of the following iteration. For PR with pre-specified number of iterations, outputs are never read back from HDFS in the case of no failures.

Skipping writing iteration output to HDFS can significantly improve the performance. However, it has implications for the implementation to retain the fault tolerance model of iHadoop. Checkpointing the output of the computation to HDFS every n iterations is one way to speed up an iterative application running time, mitigating the negative effect on fault tolerance. The experiments show that for PR application, skipping writing to HDFS makes an iteration of $\texttt{iHadoop}_a$ 15% faster on average. This reduces the normalized average iteration time of $\texttt{iHadoop}_a$ presented in Figure VI.1 from 0.79 to 0.67, achieving a speed up of 1.49 over `Hadoop`. In $\texttt{iHadoop}_{ac}$ the reduction reaches 31% per iteration. This reduces the normalized average execution from 0.62 to 0.43, achieving a speed up of 2.38 over `Hadoop`. Modifications are required to roll back iterative computations to the last saved state (i.e., the last committed output to HDFS) if a task fails. This kind of deployment can be practical for clusters with high reliability, or those that wish to minimize disk space usage or disk power consumption.

`iHadoop` applies the optimizations presented in this work to iterative applications only. We ran several non-iterative applications (e.g., word count, Pi Estimator, and grep) using `iHadoop` and compared the performance to `Hadoop`. The overhead of our implementation on non-iterative algorithms is negligible.

Chapter VII

Conclusion and Future Directions

MapReduce does not support iterative algorithms efficiently. Dataflow and task scheduling are unaware, and thus do not take advantage of, the structured behaviour of iterative algorithms. iHadoop optimizes for iterative algorithms by modifying the dataflow techniques and task scheduling to allow iterations to run asynchronously. The MapReduce framework takes advantage of a large scale partitioned parallelism as it divides the input data into many input splits that are processed in parallel. Asynchronous iterations achieve significant performance gains since they introduce more parallelism on top of that already exploited by MapReduce; namely, they allow more than one iteration to run concurrently. iHadoop runs iterations asynchronously, issues the termination check concurrently with the subsequent iteration, and schedules tasks that have direct data dependency together on the same physical machine while maintaining the strong fault tolerance mechanism and efficient load balancing of MapReduce.

VII.1 Conclusion

With typical MapReduce, an iteration I_{K+1} will have to wait until the preceding iteration I_k has finished writing its output to the DFS. This work argues that this

waiting time is not necessary; an iteration I_{k+1} can start asynchronously with an earlier iteration I_k by handling the data dependency, if it exists. The mappers of I_{k+1} can read the output of the reducers of I_k as they progress. This saves the time required to commit the output of I_k to the DFS and the I/O required to read it from there when I_{k+1} starts.

If a termination check is required after an iteration, iHadoop speculates that the check will fail and runs this check concurrently with the subsequent iteration. This speculation is correct for all the iterations required by the iterative application except for the last one. While this speculation causes the framework to perform some wasted computation, it helps starting every iteration asynchronously as early as possible.

The iHadoop model does not conflict with other important optimizations introduced in the literature, such as caching invariant data. An as shown in VI, enabling all the available optimizations results in the best performance.

VII.2 Future Work

MapReduce was introduced recently and there are still many potential improvements. Systems that compile higher-level declarative languages into MapReduce jobs can benefit from applying the techniques presented in this work to their MapReduce pipeline. For example, joining multiple tables in succession can be made efficient by streaming the output of a MapReduce join to the next; the first MapReduce job joins the first two tables and streams its output to the second MapReduce job which, then, joins the result of the first join with a third table, and so on. Similar techniques can be applied to optimize the execution of successive MapReduce jobs in systems such as Hive or Pig.

Other extensions to this work include supporting caching invariant data in map tasks. For the moment, $\texttt{iHadoop}_{ac}$ supports only caching reducers' inputs based on

the HaLoop technique which would require changes to the current task scheduler to handle the static mapping of map tasks to cluster nodes.

As shown in Subsection VI.3.4, the number of linked mappers per reducer L affects the performance of asynchronous iterations. This number is predefined for each iterative job. However, iHadoop can take advantage of the iterative job's history to optimize the execution of the following iterations, by trying several values of L at the beginning of the application's run and choosing the one that results in the best performance.

There are more experiments that are needed to fully understand the performance gains that iHadoop achieves over Hadoop for iterative applications, including the fault tolerance and load balancing. We are planning to run more experiments to test the effectiveness; given the same amount of time, how the output of iHadoop is close to the required result compared to the output of Hadoop.

REFERENCES

[1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment*, 2:922–933, August 2009.

[2] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: a Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, pages 1–14, New York, NY, USA, 2009. Association for Computing Machinery Press.

[3] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proceedings of the seventh international conference on World Wide Web (WWW)*, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[4] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient Iterative Data Processing on Large Clusters. *Proceedings of the VLDB Endowment*, 3:285–296, September 2010.

[5] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference*

on Programming Language Design and Implementation (PLDI), pages 363–375, New York, NY, USA, 2010. Association for Computing Machinery Press.

[6] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-Mapreduce: Optimizing Resource Usages of Data-Parallel Applications on Multicore with Tiling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 523–534, New York, NY, USA, 2010. Association for Computing Machinery Press.

[7] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In *Neural Information Processing Systems (NIPS)*, 2007.

[8] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proceedings of the VLDB Endowment*, 3:48–57, September 2010.

[9] Grzegorz Czajkowski. Sorting 1PB with MapReduce. `http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html`, September 2008.

[10] Marc de Kruijf and Karthikeyan Sankaralingam. MapReduce for the Cell B.E. Architecture. Technical Report TR1625, Department of Computer Sciences, The University of Wisconsin-Madison, Madison, WI, 2007.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, Berkeley, CA, USA, December 2004. USENIX Association.

[12] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA - A High Performance

Dataflow Database Machine. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB)*, pages 228–237, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[13] Jaliya Ekanayake, Thilina Gunarathne, Geoffrey Fox, Atilla Soner Balkir, Christophe Poulain, Nelson Araujo, and Roger Barga. Dryadlinq for Scientific Analyses. In *Proceedings of the Fifth IEEE International Conference on e-Science (E-SCIENCE)*, pages 329–336, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a Runtime for Iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, pages 810–818, New York, NY, USA, 2010. Association for Computing Machinery Press.

[15] John F. Gantz. The Diverse and Exploding Digital Universe, March 2008.

[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP)*, pages 29–43, New York, NY, USA, 2003. Association for Computing Machinery Press.

[17] Amol Ghoting and Konstantin Makarychev. Serial and Parallel Methods for I/O Efficient Suffix Tree Construction. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data (SIGMOD)*, pages 827–840, New York, NY, USA, 2009. Association for Computing Machinery Press.

[18] Hadoop. `http://hadoop.apache.org/`, May 2011.

[19] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a MapReduce Framework on Graphics Processors. In *Proceedings*

of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 260–269, New York, NY, USA, 2008. Association for Computing Machinery Press.

[20] Benjamin Hindman, Andy Konwinski, Matei Zaharia, and Ion Stoica. A Common Substrate for Cluster Computing. In *Proceedings of the 1st USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, Berkeley, CA, USA, 2009. USENIX Association.

[21] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 59–72, New York, NY, USA, 2007. Association for Computing Machinery Press.

[22] Matthias Jarke and Jurgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16:111–152, June 1984.

[23] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the Ninth IEEE International Conference on Data Mining (ICDM)*, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[24] Jon M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 46:604–632, September 1999.

[25] Jimmy Lin, Donald Metzler, Tamer Elsayed, and Lidan Wang. Of Ivory and Smurfs: Loxodontan MapReduce Experiments for Web Search. In *Proceedings of the 18th Text REtrieval Conference (TREC)*, 2009.

[26] Jimmy Lin and Michael Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce. In *Proceedings of the Eighth Workshop on Mining and Learning*

with Graphs (MLG), pages 78–85, New York, NY, USA, 2010. Association for Computing Machinery Press.

[27] LSST. `http://www.lsst.org/lsst`, May 2011.

[28] Mahout. `http://mahout.apache.org/`, May 2011.

[29] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data (SIGMOD)*, pages 135–146, New York, NY, USA, 2010. Association for Computing Machinery Press.

[30] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. A Peer-to-Peer Framework for Supporting MapReduce Applications in Dynamic Cloud Environments. In *Cloud Computing: Principles, Systems and Applications*, chapter 7, pages 113–125. Springer, New York, NY, USA, 2010. ISBN 978-1-84996-240-7.

[31] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: a Universal Execution Engine for Distributed Data-Flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Berkeley, CA, USA, 2011. USENIX Association.

[32] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: a Not-So-Foreign Language for Data Processing. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1099–1110, New York, NY, USA, 2008. Association for Computing Machinery Press.

[33] Spiros Papadimitriou and Jimeng Sun. DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining. In *Pro-*

ceedings of the Eighth IEEE International Conference on Data Mining (ICDM), pages 512–521, Washington, DC, USA, 2008. IEEE Computer Society.

[34] Erhard Rahm and Robert Marek. Dynamic Multi-Resource Load Balancing in Parallel Database Systems. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, pages 395–406, San Francisco, CA, USA, September 1995. Morgan Kaufmann Publishers Inc.

[35] Sayan Ranu and Ambuj K. Singh. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 844–855, Washington, DC, USA, 2009. IEEE Computer Society.

[36] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data (SIGMOD)*, pages 23–34, New York, NY, USA, 1979. Association for Computing Machinery Press.

[37] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. FPMR: MapReduce Framework on FPGA. In *Proceedings of the 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 93–102, New York, NY, USA, 2010. Association for Computing Machinery Press.

[38] Nisheeth Shrivastava, Anirban Majumder, and Rajeev Rastogi. Mining (Social) Network Graphs to Detect Random Link Attacks. In *Proceedings of the IEEE 24th International Conference on Data Engineering (ICDE)*, pages 486–495, Washington, DC, USA, 2008. IEEE Computer Society.

[39] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A Wide-Area Distributed Database System. *The VLDB Journal*, 5:048–063, January 1996.

[40] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2:1626–1629, August 2009.

[41] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1013–1020, New York, NY, USA, 2010. Association for Computing Machinery Press.

[42] Abhishek Verma, Xavier Llorà, David E. Goldberg, and Roy H. Campbell. Scaling Genetic Algorithms Using MapReduce. In *Proceedings of the Ninth International Conference on Intelligent Systems Design and Applications (ISDA)*, pages 13–18, Washington, DC, USA, 2009. IEEE Computer Society.

[43] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing Cloud Computing Hardware Reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 193–204, New York, NY, USA, 2010. Association for Computing Machinery Press.

[44] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *Proceedings of the ACM/SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1029–1040, New York, NY, USA, 2007. Association for Computing Machinery Press.

[45] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society.

[46] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: a System for General-Purpose Distributed Data-Parallel Computing using a High-Level Language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, Berkeley, CA, USA, 2008. USENIX Association.

[47] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, Berkeley, CA, USA, 2010. USENIX Association.

[48] Yanfeng Zhang, Qinxin Gao, Lixin Gao, and Cuirong Wang. iMapReduce: A Distributed Computing Framework for Iterative Computation. In *Proceedings of the 1st International Workshop on Data Intensive Computing in the Clouds*, 2011.

[49] Feida Zhu, Xifeng Yan, Jiawei Han, and Philip S. Yu. gPrune: a Constraint Pushing Framework for Graph Pattern Mining. In *Proceedings of the 11th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD)*, pages 388–400, Berlin, Heidelberg, 2007. Springer-Verlag.

APPENDIX A

iHadoop API

This appendix lists and describes the APIs of the different systems related to this work. As we mentioned in Chapter V, iHadoop builds on top of HaLoop, and thus, its API extends what is provided by HaLoop. Hadoop programs can run without modifications on HaLoop and iHadoop. However, to run an iterative MapReduce program with HaLoop or iHadoop, we have to provide some information to the runtime system through its API. We first describe the Hadoop API and its usage. Then, we discuss the API extensions HaLoop presented to provide iterative programming support for iterative applications. Finally, we present the iHadoop extensions and changes. Please note that we are not presenting a full list for the API in this appendix, the interested reader can refer to the source code for a complete API list.

A.1 Hadoop API

Hadoop provides a programming support that is sufficient to define a MapReduce job and tune it. For every MapReduce job, the user has to define the *map* and the *reduce* functions to be used in the job. This appendix summarizes the Hadoop API in the following.

- `setMapperClass`: directs the runtime to the implementation of the *map* function. This is required for every MapReduce job.
- `setReducerClass`: directs the runtime to the implementation of the *reduce* function. This is required for every MapReduce job.
- `setNumReduceTasks`: specifies the number of reduce tasks to be launched for the job. However, `setNumMapTasks` is used as a hint since the number of map tasks is determined according to the size of the input.
- `setOutputKeyClass`: specifies the key type of the output.
- `setOutputValueClass`: specifies the value type of the output.
- `setInputFormat`: directs the runtime to a handler for the input format. The handler will be used to partition the input into splits and to obtain an input stream with record granularity.
- `setOutputFormat`: directs the runtime to a handler for the output format. The handler will be used to obtain an output stream with record granularity and to commit the output to HDFS.
- `setInputPaths`: specifies the input paths for the job. The input paths should be valid directories in HDFS.
- `setOutputPath`: specifies the output path for the job. The output paths should be a valid directory in HDFS.
- `submitJob`: submits a job to the runtime. The job information is passed through a wrapper object; `JobConf`. `JobConf` contains all the job configuration options and is distributed across the nodes of the cluster.

A.2 HaLoop API

- `setIterative`: is used whenever submitting an iterative job to HaLoop. This determines the behaviour of the `JobTracker` and the `TaskScheduler`. If the job is not iterative, the behaviour should be identical to Hadoop. Otherwise, HaLoop optimizations, e.g., caching invariant data and loop-aware scheduling, are set to action.

- `setStepConf`: is used to define the required steps in every iteration of the algorithm. As we will show in Appendix B, every iteration of PageRank has two steps: a rank aggregation step and a join step; every iteration of Descendant Query has two steps: a join step and a duplicate elimination step. `setStepConf` defines a `JobConf` for every step in the iterative application.

- `setLoopInputOutput`: specifies a handler for the application input and output. The runtime queries this handler to determine the input paths and the output path for every step throughout the execution of the iterative application.

- `setNumIterations`: sets a limit on the number of iterations the iterative application can run.

- `setLoopMapCacheSwitch`: specifies a handler which is queried by a map task of an iterative application to determine whether (1) it should cache its input or (2) it should read cached data as input.

- `setLoopMapCacheFilter`: determines which part of the input data of a map task to be cached.

- `setLoopReduceCacheSwitch`: specifies a handler which is queried by a reduce task of an iterative application to determine whether (1) it should cache its input or (2) it should read cached data as input.

- `setLoopReduceCacheFilter`: determines which part of the input data of a reduce task to be cached.

A.3 iHadoop API

- `setAsynchronousIterations`: is set to `TRUE` if we need to activate the asynchronous iterations support during the execution of the iterative application. It changes the behaviour of the `JobTracker` and the `TaskScheduler` to be aware of the iterations that are running asynchronously.

- `setEarlyStart`: activates the reducer early start (as discussed in V.1). It is set to `TRUE`, e.g., *on*, by default.

- `setLinkedNum`: specifies the number of linked map tasks per reduce (as discussed in V.2). It is set to 3 by default.

- `setMapJVMReuse`: enables/disables the reuse of the linked map JVM to mitigate the overhead of creating an socket stream between a reducer and a mapper of the following iteration. It is set to `TRUE`, e.g., *on*, by default.

- `setTCControl`: specifies a handler to the termination check of an iterative algorithm. This handler specifies the following: (1) if the application needs to run a termination check after a specific step/iteration, (2) obtain a `JobConf` for the termination condition where the input paths and the output path have been adjusted, and (3) if the overall computation needs to be terminated whenever the termination condition is found to be satisfied after the execution of the termination check.

- `setHDFSInputs`: specifies a handler for the application input and output. This works in tandem with `setLoopInputOutput`. Since the input to an iteration can be a combination of the output of the previous iteration and some data

from the HDFS, this handler determines which data are to be streamed from the previous asynchronously-running iteration and which data are to be read from HDFS.

- `setReduceBufferSize`: specifies the size of the buffer associated with the output stream of a reducer that is streaming its output to one or more map tasks of the following iteration. It is set to 8192 bytes by default.

- `setMapBufferSize`: specifies the size of the buffer associated with the input stream of a mapper that is reading in its input from a reducer of the previous iteration. It is set to 2048 bytes by default.

APPENDIX B

Iterative Algorithms

This appendix will go into the details of the algorithms used in the experiments section of this work, namely PageRank and Descendant Query. The appendix will give an overview for each and present its implementation with MapReduce. Finally, I will include how this application can be run under Hadoop, HaLoop, and iHadoop.

B.1 PageRank

PageRank is a link analysis algorithm and was first introduced in 1998 by Brin and Page [3]. It represents the web as a large graph where each web page is a vertex and every outbound link from a web page to another is an edge. It ranks web pages by iteratively updating the weight of each vertex based on the weights of all other web pages that have a direct inbound link to it. Tables B.1(a)–B.1(d) show an example of the tables to be used for PageRank.

Each iteration of PageRank can be represented as two MapReduce jobs. The first job (step) is a join operation that would join the ranking table, which is a list of every web page and its corresponding rank, with the linkage table, which is a list of every edge in the graph. The mappers of this step read the two tables and emit a record using `url` as a key. The reducers actually perform the join for every `url`, i.e., key.

The output of this step is the rank adjustment for outbound links for every web page. The second step is a rank aggregation operation that sums the rank for each `url` in the rank adjustment table. The mappers of this step emit records with the `url` as a key and the `rank` as value. The reducers aggregate the values for every key and produce the new rank. The output of the reducers is used as the new ranking table for the join operation of the next step.

`url`	`rank`
www.bing.com	1.000
www.google.com	1.000
www.msn.com	1.000
www.yahoo.com	1.000

(a) Initial rank table

`url`	`dest_url`
www.bing.com	www.google.com
www.yahoo.com	www.google.com
www.msn.com	www.yahoo.com
www.msn.com	www.google.com
www.bing.com	www.msn.com
www.bing.com	www.yahoo.com

(b) Linkage table

`url`	`rank_cont`
www.google.com	0.334
www.msn.com	0.334
www.yahoo.com	0.334
www.bing.com	0.176
www.yahoo.com	0.500
www.google.com	0.500
www.msn.com	0.176
www.google.com	1.000
www.yahoo.com	0.176

(c) Rank contributions

`url`	`rank`
www.bing.com	0.176
www.google.com	1.834
www.msn.com	0.509
www.yahoo.com	1.009

(d) New rank table

Table B.1: PageRank example.

B.1.1 Termination Condition

PageRank can run for a predefined number of iterations, or until it converges. Convergence in PageRank happens when the ranks assigned to web pages are very much the same between two consecutive iterations.

Assume that we have a web dataset of $COUNT$ pages. Then, for every web page p^i in the dataset where $0 < i < COUNT$, an iteration I_k assigns a ranking value p^i_k.

And the next iteration I_{k+1} assigns the same web page p^i the value p^i_{k+1}. PageRank reaches convergence when the condition $\sum_{i=0}^{COUNT} \|p^i_{k+1} - p^i_k\| < threshold$ is satisfied, where *threshold* is positive value that has to be set carefully; a large *threshold* value can hinder the quality of the ranks assigned with PageRank, and a small *threshold* can cause the application to run so many iterations.

B.2 Descendant Query

Descendant Query is used in social networks to find (in)direct relations between entities. Many social networks, e.g., Facebook, make suggests to their users based on "*friends of friends*" relation. Descendant query represents a social network as a graph: every entity is a vertex, and a relation between two entities is an edge linking the respective vertices. Descendant Query finds those vertices that are within n hops/edges away. Tables B.2(a)–B.2(d) show an example of the tables to be used for Descendant Query.

Each iteration of Descendant Query can be represented as two MapReduce jobs. The first job is a join operation that would join the relation table `R`, which is a list of every entity and its related entities, with the entities discovered in the earlier iteration `F` using the predicate `R.name = F.friend`. The mappers of this step read the two tables and emit a record using `name` as a key. The reducers actually perform the join. The output of this step is the rank adjustment for outbound links for every web page. The second step is a duplicate elimination operation to remove the entities discovered from earlier iterations (to avoid circles). The output of this step is used as the input F for the following iteration.

name	friend
Mendel Rosenblum	Ken Thompson
Andrew Tanenbaum	Gordon Moore
Gordon Moore	Robert Floyd
Linus Torvalds	Edsger Dijkstra
Ken Thompson	Gordon Moore
David Huffman	John von Neumann
Gordon Moore	Dennis Ritchie
Ken Thompson	Richard Stallman
Marshall McKusick	Edsger Dijkstra
John von Neumann	Alfred Aho
Richard Stallman	David Huffman

(a) Relation table R

name	friend
Mendel Rosenblum	Ken Thompson

(b) F_0

name	friend
Mendel Rosenblum	Gordon Moore
Mendel Rosenblum	Richard Stallman

(c) F_1

name	friend
Mendel Rosenblum	Robert Floyd
Mendel Rosenblum	Dennis Ritchie
Mendel Rosenblum	David Huffman

(d) F_2

name	friend
Mendel Rosenblum	John von Neumann

(e) F_3

Table B.2: Descendant Query example.

B.3 Parallel Breadth-First Search

Parallel Breadth-First Search (PBFS) is one of the common search algorithms for large graphs. It is used to construct a *tree* of the nodes that can be reached from a certain source. For example, it can be used to determine whether some states are reachable from the current state or not.

Each iteration of PBFS is represented by a single MapReduce job. The mappers read the adjacency list of the graph, and then, they mark and update the distance of the nodes that can be reached through the key input. The reducers of PBFS emit the updated version of the adjacency list by choosing the least distance value discovered for each node.

Tables B.3(a)-B.3(d) show an example of the tables to be used for PBFS.

src	dest	state	cost
1	2, 4, 6	READY	0
2	1, 5, 6	-	∞
3	4, 6	-	∞
4	1, 3, 5	-	∞
5	2, 4	-	∞
6	1, 2, 3	-	∞

(a) Initial adjacency list

src	dest	state	cost
1	2, 4, 6	DONE	0
2	1, 5, 6	READY	1
3	4, 6	-	∞
4	1, 3, 5	READY	1
5	2, 4	-	∞
6	1, 2, 3	READY	1

(b) After first iteration

src	dest	state	cost
1	2, 4, 6	DONE	0
2	1, 5, 6	DONE	1
3	4, 6	READY	2
4	1, 3, 5	DONE	1
5	2, 4	READY	2
6	1, 2, 3	DONE	1

(c) After second iteration

src	dest	state	cost
1	2, 4, 6	DONE	0
2	1, 5, 6	DONE	1
3	4, 6	DONE	2
4	1, 3, 5	DONE	1
5	2, 4	DONE	2
6	1, 2, 3	DONE	1

(d) After third iteration

Table B.3: Parallel breadth-first search example